# Table of contents

## 1 Get Started
1.1 Introduction ................................................................. 3
1.2 What You Need ............................................................... 3
  1.2.1 Hardware ............................................................... 3
  1.2.2 Software ............................................................. 53
1.3 Installation ................................................................. 53
  1.3.1 IDE ................................................................. 54
  1.3.2 Manual Installation .................................................. 54
1.4 Build Your First Project .................................................. 86
1.5 Uninstall ESP-IDF .......................................................... 86

## 2 API Reference
2.1 API Conventions ............................................................ 87
  2.1.1 Error Handling ....................................................... 87
  2.1.2 Configuration Structures .......................................... 87
  2.1.3 Private APIs .......................................................... 89
  2.1.4 Components in Example Projects ................................... 89
  2.1.5 API Stability .......................................................... 89
2.2 Application Protocols ..................................................... 90
  2.2.1 ASIO Port ............................................................ 90
  2.2.2 ESP-Modbus ......................................................... 91
  2.2.3 ESP-MQTT ............................................................ 91
  2.2.4 ESP-TLS .............................................................. 109
  2.2.5 ESP HTTP Client ..................................................... 129
  2.2.6 ESP Local Control .................................................... 146
  2.2.7 ESP Serial Slave Link .............................................. 156
  2.2.8 ESP x509 Certificate Bundle ...................................... 170
  2.2.9 HTTP Server ........................................................ 173
  2.2.10 HTTPS Server ....................................................... 201
  2.2.11 ICMP Echo ........................................................ 205
  2.2.12 mDNS Service ....................................................... 210
  2.2.13 Mbed TLS .......................................................... 211
  2.2.14 IP Network Layer .................................................. 213
2.3 Error Codes Reference .................................................... 213
2.4 Networking APIs .......................................................... 219
  2.4.1 Wi-Fi ............................................................... 219
  2.4.2 Ethernet ............................................................. 356
  2.4.3 Thread ............................................................... 390
  2.4.4 ESP-NETIF .......................................................... 399
  2.4.5 IP Network Layer .................................................... 436
  2.4.6 Application Layer .................................................... 439
2.5 Peripherals API ............................................................ 439
  2.5.1 Analog to Digital Converter (ADC) Oneshot Mode Driver .... 439
  2.5.2 Analog to Digital Converter (ADC) Continuous Mode Driver ... 450
  2.5.3 Analog to Digital Converter (ADC) Calibration Driver ....... 460
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.4</td>
<td>Clock Tree</td>
<td>463</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Digital To Analog Converter (DAC)</td>
<td>474</td>
</tr>
<tr>
<td>2.5.6</td>
<td>GPIO &amp; RTC GPIO</td>
<td>488</td>
</tr>
<tr>
<td>2.5.7</td>
<td>General Purpose Timer (GPTimer)</td>
<td>510</td>
</tr>
<tr>
<td>2.5.8</td>
<td>Dedicated GPIO</td>
<td>525</td>
</tr>
<tr>
<td>2.5.9</td>
<td>Hash-Based Message Authentication Code (HMAC)</td>
<td>531</td>
</tr>
<tr>
<td>2.5.10</td>
<td>Digital Signature (DS)</td>
<td>536</td>
</tr>
<tr>
<td>2.5.11</td>
<td>Inter-Integrated Circuit (I2C)</td>
<td>541</td>
</tr>
<tr>
<td>2.5.12</td>
<td>Inter-IC Sound (I2S)</td>
<td>563</td>
</tr>
<tr>
<td>2.5.13</td>
<td>LCD</td>
<td>588</td>
</tr>
<tr>
<td>2.5.14</td>
<td>LED Control (LED)</td>
<td>606</td>
</tr>
<tr>
<td>2.5.15</td>
<td>Pulse Counter (PCNT)</td>
<td>626</td>
</tr>
<tr>
<td>2.5.16</td>
<td>Remote Control Transceiver (RMT)</td>
<td>641</td>
</tr>
<tr>
<td>2.5.17</td>
<td>SD SPI Host Driver</td>
<td>668</td>
</tr>
<tr>
<td>2.5.18</td>
<td>Sigma-Delta Modulation (SDM)</td>
<td>674</td>
</tr>
<tr>
<td>2.5.19</td>
<td>SPI Flash API</td>
<td>680</td>
</tr>
<tr>
<td>2.5.20</td>
<td>SPI Master Driver</td>
<td>711</td>
</tr>
<tr>
<td>2.5.21</td>
<td>SPI Slave Driver</td>
<td>734</td>
</tr>
<tr>
<td>2.5.22</td>
<td>SPI Slave Half Duplex</td>
<td>741</td>
</tr>
<tr>
<td>2.5.23</td>
<td>Temperature Sensor</td>
<td>749</td>
</tr>
<tr>
<td>2.5.24</td>
<td>Touch Sensor</td>
<td>753</td>
</tr>
<tr>
<td>2.5.25</td>
<td>Touch Element</td>
<td>782</td>
</tr>
<tr>
<td>2.5.26</td>
<td>Two-Wire Automotive Interface (TWAI)</td>
<td>812</td>
</tr>
<tr>
<td>2.5.27</td>
<td>Universal Asynchronous Receiver/Transmitter (UART)</td>
<td>835</td>
</tr>
<tr>
<td>2.5.28</td>
<td>USB Device Driver</td>
<td>860</td>
</tr>
<tr>
<td>2.5.29</td>
<td>USB Host</td>
<td>864</td>
</tr>
<tr>
<td>2.6</td>
<td>Project Configuration</td>
<td>911</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Introduction</td>
<td>911</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Project Configuration Menu</td>
<td>911</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Using sdkconfig.defaults</td>
<td>912</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Kconfig Format Rules</td>
<td>912</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Backward Compatibility of Kconfig Options</td>
<td>912</td>
</tr>
<tr>
<td>2.6.6</td>
<td>Configuration Options Reference</td>
<td>913</td>
</tr>
<tr>
<td>2.7</td>
<td>Provisioning API</td>
<td>1223</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Protocol Communication</td>
<td>1223</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Unified Provisioning</td>
<td>1239</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Wi-Fi Provisioning</td>
<td>1246</td>
</tr>
<tr>
<td>2.8</td>
<td>Storage API</td>
<td>1265</td>
</tr>
<tr>
<td>2.8.1</td>
<td>FAT Filesystem Support</td>
<td>1266</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Manufacturing Utility</td>
<td>1275</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Non-Volatile Storage Library</td>
<td>1280</td>
</tr>
<tr>
<td>2.8.4</td>
<td>NVS Encryption</td>
<td>1303</td>
</tr>
<tr>
<td>2.8.5</td>
<td>NVS Partition Generator Utility</td>
<td>1309</td>
</tr>
<tr>
<td>2.8.6</td>
<td>NVS Partition Parser Utility</td>
<td>1315</td>
</tr>
<tr>
<td>2.8.7</td>
<td>SD/SDIO/MMC Driver</td>
<td>1316</td>
</tr>
<tr>
<td>2.8.8</td>
<td>Partitions API</td>
<td>1322</td>
</tr>
<tr>
<td>2.8.9</td>
<td>SPIFFS Filesystem</td>
<td>1332</td>
</tr>
<tr>
<td>2.8.10</td>
<td>Virtual Filesystem Component</td>
<td>1336</td>
</tr>
<tr>
<td>2.8.11</td>
<td>Wear Levelling API</td>
<td>1352</td>
</tr>
<tr>
<td>2.9</td>
<td>System API</td>
<td>1355</td>
</tr>
<tr>
<td>2.9.1</td>
<td>App Image Format</td>
<td>1355</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Bootloader Image Format</td>
<td>1361</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Application Level Tracing</td>
<td>1363</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Call Function with External Stack</td>
<td>1369</td>
</tr>
<tr>
<td>2.9.5</td>
<td>Chip Revision</td>
<td>1370</td>
</tr>
<tr>
<td>2.9.6</td>
<td>Console</td>
<td>1374</td>
</tr>
<tr>
<td>2.9.7</td>
<td>eFuse Manager</td>
<td>1383</td>
</tr>
<tr>
<td>2.9.8</td>
<td>Error Code and Helper Functions</td>
<td>1413</td>
</tr>
</tbody>
</table>
3 Hardware Reference

4 API Guides

4.1 Application Level Tracing Library
   4.1.1 Overview
   4.1.2 Modes of Operation
   4.1.3 Configuration Options and Dependencies
   4.1.4 How to Use This Library
4.2 Application Startup Flow
   4.2.1 First Stage Bootloader
   4.2.2 Second Stage Bootloader
   4.2.3 Application Startup
4.3 Bootloader
   4.3.1 Bootloader Compatibility
   4.3.2 Log Level
   4.3.3 Factory Reset
   4.3.4 Boot from Test Firmware
   4.3.5 Rollback
   4.3.6 Watchdog
   4.3.7 Bootloader Size
   4.3.8 Fast Boot from Deep-Sleep
   4.3.9 Custom Bootloader
4.4 Build System
   4.4.1 Overview
   4.4.2 Using the Build System
   4.4.3 Example Project
   4.4.4 Project CMakeLists File
   4.4.5 Component CMakeLists Files
   4.4.6 Component Configuration
   4.4.7 Preprocessor Definitions
   4.4.8 Component Requirements
4.11 Error Handling ................................. 1851
4.11.1 Overview .................................... 1851

4.10 Device Firmware Upgrade via USB ............ 1830
4.10.1 USB Connection .............................. 1830
4.10.2 Building the DFU Image ...................... 1830
4.10.3 Flashing the DFU Image ...................... 1830
4.10.4 Udev Rule (Linux Only) ...................... 1830
4.10.5 USB Drivers (Windows Only) ................ 1830
4.10.6 Common Errors and Known Issues .......... 1830
4.10.7 Secure Download Mode ........................ 1830

4.9 Deep Sleep Wake Stubs ......................... 1828
4.9.1 Rules for Wake Stubs ........................ 1828
4.9.2 Implementing A Stub .......................... 1828
4.9.3 Loading Code Into RTC Memory .............. 1828
4.9.4 Loading Data Into RTC Memory .............. 1828
4.9.5 CRCCheck For Wake Stubs ..................... 1828
4.9.6 Example ..................................... 1828

4.8 Current Consumption Measurement of Modules .. 1826
4.8.1 Notes to Measurement .......................... 1826
4.8.2 Hardware Connection .......................... 1826
4.8.3 Measurement Steps ............................. 1826

4.7 Core Dump ...................................... 1824
4.7.1 Overview ..................................... 1824
4.7.2 Configurations ................................ 1824
4.7.3 Core Dump to Flash ............................ 1824
4.7.4 Core Dump to UART ............................ 1824
4.7.5 Core Dump Commands ........................... 1824
4.7.6 ROM Functions in Backtraces ................. 1824
4.7.7 Dumping Variables on Demand ............... 1824
4.7.8 Running idf.py coredump-info and idf.py coredump-debug ............ 1824

4.6 C++ Support .................................... 1822
4.6.1 esp-idf-cxx Component ....................... 1822
4.6.2 C++ Language Standard ....................... 1822
4.6.3 Multithreading ................................ 1822
4.6.4 Exception Handling ............................ 1822
4.6.5 Runtime Type Information (RTTI) ............. 1822
4.6.6 Developing in C++ ............................. 1822
4.6.7 Limitations ................................... 1822
4.6.8 What to Avoid ................................ 1822

4.5 C Support ...................................... 1820
4.5.1 C Version .................................... 1820
4.5.2 Unsupported C Features ........................ 1820

4.4 Setting Up ESP-IDF ................................ 1818
4.4.1 ESP-IDF Build System API ..................... 1818
4.4.2 ESP-IDFmake ................................. 1818
4.4.3 idf.py ..................................... 1818
4.4.4 Component ................................. 1818

4.3 ESP-IDF Project .................................. 1815
4.3.1 Project Directory Structure ................. 1815
4.3.2 Example Project Structure .................... 1815
4.3.3 ESP-IDF Project Variables .................... 1815
4.3.4 Dependencies ................................ 1815

4.2 Introduction .................................... 1813
4.2.1 How to Access the ESP-IDF User Manual .... 1813
4.2.2 Project Structure .............................. 1813
4.2.3 Project Variables ............................. 1813
4.2.4 ESP-IDF Project Dependencies ............... 1813

4 System Overview ................................ 1810
4.1 ESP-IDF Overview ................................ 1810
4.1.1 Components .................................. 1810
4.1.2 Libraries ...................................... 1810
4.1.3 Common Components ............................ 1810

4.1 System Overview ................................ 1808
4.1.1 ESP-IDF Overview .............................. 1808
4.1.2 ESP-IDF Components ............................ 1808
4.1.3 ESP-IDF Libraries .............................. 1808
4.1.4 Common ESP-IDF Components ................. 1808

4 ESP-IDF Overview ................................. 1806
4.1 ESP-IDF Components .............................. 1806
4.2 ESP-IDF Libraries ................................. 1806
4.3 Common ESP-IDF Components ..................... 1806
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.19</td>
<td>lwIP</td>
</tr>
<tr>
<td>4.19.1</td>
<td>Supported APIs</td>
</tr>
<tr>
<td>4.19.2</td>
<td>BSD Sockets API</td>
</tr>
<tr>
<td>4.19.3</td>
<td>Netconn API</td>
</tr>
<tr>
<td>4.19.4</td>
<td>lwIP FreeRTOS Task</td>
</tr>
<tr>
<td>4.19.5</td>
<td>IPv6 Support</td>
</tr>
<tr>
<td>4.19.6</td>
<td>ESP-lwIP Custom Modifications</td>
</tr>
<tr>
<td>4.19.7</td>
<td>Performance Optimization</td>
</tr>
<tr>
<td>4.20</td>
<td>Memory Types</td>
</tr>
<tr>
<td>4.20.1</td>
<td>DRAM (Data RAM)</td>
</tr>
<tr>
<td>4.20.2</td>
<td>IRAM (Instruction RAM)</td>
</tr>
<tr>
<td>4.20.3</td>
<td>IROM (Code Executed from flash)</td>
</tr>
<tr>
<td>4.20.4</td>
<td>DROM (Data Stored in flash)</td>
</tr>
<tr>
<td>4.20.5</td>
<td>RTC Slow Memory</td>
</tr>
<tr>
<td>4.20.6</td>
<td>RTC FAST Memory</td>
</tr>
<tr>
<td>4.20.7</td>
<td>DMA-Capable Requirement</td>
</tr>
<tr>
<td>4.20.8</td>
<td>DMA Buffer in the Stack</td>
</tr>
<tr>
<td>4.21</td>
<td>OpenThread</td>
</tr>
<tr>
<td>4.21.1</td>
<td>Modes of the OpenThread Stack</td>
</tr>
<tr>
<td>4.21.2</td>
<td>How to Write an OpenThread Application</td>
</tr>
<tr>
<td>4.21.3</td>
<td>The OpenThread Border Router</td>
</tr>
<tr>
<td>4.22</td>
<td>Partition Tables</td>
</tr>
<tr>
<td>4.22.1</td>
<td>Overview</td>
</tr>
<tr>
<td>4.22.2</td>
<td>Built-in Partition Tables</td>
</tr>
<tr>
<td>4.22.3</td>
<td>Creating Custom Tables</td>
</tr>
<tr>
<td>4.22.4</td>
<td>Generating Binary Partition Table</td>
</tr>
<tr>
<td>4.22.5</td>
<td>Partition Size Checks</td>
</tr>
<tr>
<td>4.22.6</td>
<td>Flashing the Partition Table</td>
</tr>
<tr>
<td>4.22.7</td>
<td>Partition Tool (parttool.py)</td>
</tr>
<tr>
<td>4.23</td>
<td>Performance</td>
</tr>
<tr>
<td>4.23.1</td>
<td>How to Optimize Performance</td>
</tr>
<tr>
<td>4.23.2</td>
<td>Guides</td>
</tr>
<tr>
<td>4.24</td>
<td>Reproducible Builds</td>
</tr>
<tr>
<td>4.24.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.24.2</td>
<td>Reasons for Non-Reproducible Builds</td>
</tr>
<tr>
<td>4.24.3</td>
<td>Enabling Reproducible Builds in ESP-IDF</td>
</tr>
<tr>
<td>4.24.4</td>
<td>How Reproducible Builds Are Achieved</td>
</tr>
<tr>
<td>4.24.5</td>
<td>Reproducible Builds and Debugging</td>
</tr>
<tr>
<td>4.24.6</td>
<td>Factors Which Still Affect Reproducible Builds</td>
</tr>
<tr>
<td>4.25</td>
<td>RF Calibration</td>
</tr>
<tr>
<td>4.25.1</td>
<td>Partial Calibration</td>
</tr>
<tr>
<td>4.25.2</td>
<td>Full Calibration</td>
</tr>
<tr>
<td>4.25.3</td>
<td>No Calibration</td>
</tr>
<tr>
<td>4.25.4</td>
<td>PHY Initialization Data</td>
</tr>
<tr>
<td>4.25.5</td>
<td>API Reference</td>
</tr>
<tr>
<td>4.26</td>
<td>Thread Local Storage</td>
</tr>
<tr>
<td>4.26.1</td>
<td>Overview</td>
</tr>
<tr>
<td>4.26.2</td>
<td>FreeRTOS Native APIs</td>
</tr>
<tr>
<td>4.26.3</td>
<td>Pthread APIs</td>
</tr>
<tr>
<td>4.26.4</td>
<td>C11 Standard</td>
</tr>
<tr>
<td>4.27</td>
<td>Tools</td>
</tr>
<tr>
<td>4.27.1</td>
<td>IDF Frontend - idf.py</td>
</tr>
<tr>
<td>4.27.2</td>
<td>IDF Docker Image</td>
</tr>
<tr>
<td>4.27.3</td>
<td>IDF Windows Installer</td>
</tr>
<tr>
<td>4.27.4</td>
<td>IDF Component Manager</td>
</tr>
<tr>
<td>4.27.5</td>
<td>IDF Clang-Tidy</td>
</tr>
<tr>
<td>4.27.6</td>
<td>Downloadable IDF Tools</td>
</tr>
<tr>
<td>4.28</td>
<td>Unit Testing in ESP32-S2</td>
</tr>
</tbody>
</table>
4.28.1 Normal Test Cases ............................................. 2020
4.28.2 Multi-device Test Cases ...................................... 2020
4.28.3 Multi-stage Test Cases ....................................... 2021
4.28.4 Tests For Different Targets ................................. 2022
4.28.5 Building Unit Test App ....................................... 2022
4.28.6 Running Unit Tests ............................................ 2023
4.28.7 Timing Code with Cache Compensated Timer .......... 2024
4.28.8 Mocks ........................................................... 2025

4.29 Running ESP-IDF Applications on Host .................... 2027
4.29.1 Introduction .................................................. 2027
4.29.2 Requirements for Using Mocks ............................. 2028
4.29.3 Build and Run ................................................ 2028
4.29.4 Component Linux/Mock Support Overview ............. 2029

4.30 USB OTG Console ................................................. 2030
4.30.1 Hardware Requirements ..................................... 2030
4.30.2 Software Configuration ....................................... 2030
4.30.3 Uploading the Application .................................. 2030
4.30.4 Limitations ........................................................ 2031

4.31 Wi-Fi Driver ........................................................ 2031
4.31.1 ESP32-S2 Wi-Fi Feature List ............................... 2031
4.31.2 How To Write a Wi-Fi Application ......................... 2032
4.31.3 ESP32-S2 Wi-Fi API Error Code ............................ 2033
4.31.4 ESP32-S2 Wi-Fi API Parameter Initialization .......... 2033
4.31.5 ESP32-S2 Wi-Fi Programming Model .................... 2033
4.31.6 ESP32-S2 Wi-Fi Event Description ....................... 2033
4.31.7 ESP32-S2 Wi-Fi Station General Scenario ............. 2037
4.31.8 ESP32-S2 Wi-Fi AP General Scenario .................. 2039
4.31.9 ESP32-S2 Wi-Fi Scan ......................................... 2041
4.31.10 ESP32-S2 Wi-Fi Station Connecting Scenario ........ 2047
4.31.11 ESP32-S2 Wi-Fi Station Connecting When Multiple APs Are Found 2054
4.31.12 Wi-Fi Reconnect ............................................. 2054
4.31.13 Wi-Fi Beacon Timeout ...................................... 2054
4.31.14 ESP32-S2 Wi-Fi Configuration ............................ 2054
4.31.15 Wi-Fi Easy Connect™ (DPP) ............................... 2060
4.31.16 Wi-Fi Aware™ (NAN) ....................................... 2061
4.31.17 Wireless Network Management ........................... 2061
4.31.18 Radio Resource Measurement .............................. 2062
4.31.19 Fast BSS Transition ....................................... 2062
4.31.20 Wi-Fi Location ................................................ 2062
4.31.21 ESP32-S2 Wi-Fi Power-saving Mode ................... 2063
4.31.22 ESP32-S2 Wi-Fi Throughput ............................... 2064
4.31.23 Wi-Fi 80211 Packet Send .................................. 2065
4.31.24 Wi-Fi Sniffer Mode ......................................... 2066
4.31.25 Wi-Fi Multiple Antennas ................................. 2067
4.31.26 Wi-Fi Channel State Information ......................... 2068
4.31.27 Wi-Fi Channel State Information Configure .......... 2070
4.31.28 Wi-Fi HT20/40 .............................................. 2070
4.31.29 Wi-Fi QoS .................................................... 2070
4.31.30 Wi-Fi AMSDU ................................................ 2071
4.31.31 Wi-Fi Fragment .............................................. 2071
4.31.32 WPS Enrollee ............................................... 2071
4.31.33 Wi-Fi Buffer Usage ........................................ 2071
4.31.34 How to Improve Wi-Fi Performance ................. 2072
4.31.35 Wi-Fi Menuconfig .......................................... 2076
4.31.36 Troubleshooting ............................................. 2078

4.32 Wi-Fi Security ...................................................... 2081
4.32.1 ESP32-S2 Wi-Fi Security Features ....................... 2081
4.32.2 Protected Management Frames (PMF) ................... 2084
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.32.3</td>
<td>Wi-Fi Enterprise</td>
<td>2085</td>
</tr>
<tr>
<td>4.32.4</td>
<td>WPA3-Personal</td>
<td>2085</td>
</tr>
<tr>
<td>4.32.5</td>
<td>Wi-Fi Enhanced Open™</td>
<td>2087</td>
</tr>
<tr>
<td>5</td>
<td>Security Guides</td>
<td>2089</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>2089</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Security</td>
<td>2093</td>
</tr>
<tr>
<td>5.2</td>
<td>Features</td>
<td>2093</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Flash Encryption</td>
<td>2108</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Secure Boot V2</td>
<td>2108</td>
</tr>
<tr>
<td>5.3</td>
<td>Workflows</td>
<td>2116</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Host-Based Security Workflows</td>
<td>2116</td>
</tr>
<tr>
<td>6</td>
<td>Migration Guides</td>
<td>2125</td>
</tr>
<tr>
<td>6.1</td>
<td>ESP-IDF 5.x Migration Guide</td>
<td>2125</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Migration from 4.4 to 5.0</td>
<td>2125</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Migration from 5.0 to 5.1</td>
<td>2153</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Migration from 5.1 to 5.2</td>
<td>2157</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Migration from 5.2 to 5.3</td>
<td>2159</td>
</tr>
<tr>
<td>7</td>
<td>Libraries and Frameworks</td>
<td>2161</td>
</tr>
<tr>
<td>7.1</td>
<td>Cloud Frameworks</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.1</td>
<td>ESP RainMaker</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.2</td>
<td>AWS IoT</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Azure IoT</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Google IoT Core</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Aliyun IoT</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.6</td>
<td>Joylink IoT</td>
<td>2161</td>
</tr>
<tr>
<td>7.1.7</td>
<td>Tencent IoT</td>
<td>2162</td>
</tr>
<tr>
<td>7.1.8</td>
<td>Tencentyun IoT</td>
<td>2162</td>
</tr>
<tr>
<td>7.1.9</td>
<td>Baidu IoT</td>
<td>2162</td>
</tr>
<tr>
<td>7.2</td>
<td>Espressif’s Frameworks</td>
<td>2162</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Espressif Audio Development Framework</td>
<td>2162</td>
</tr>
<tr>
<td>7.2.2</td>
<td>ESP-CSI</td>
<td>2162</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Espressif DSP Library</td>
<td>2162</td>
</tr>
<tr>
<td>7.2.4</td>
<td>ESP-WIFI-MESH Development Framework</td>
<td>2163</td>
</tr>
<tr>
<td>7.2.5</td>
<td>ESP-WHO</td>
<td>2163</td>
</tr>
<tr>
<td>7.2.6</td>
<td>ESP RainMaker</td>
<td>2163</td>
</tr>
<tr>
<td>7.2.7</td>
<td>ESP-IoT-Solution</td>
<td>2163</td>
</tr>
<tr>
<td>7.2.8</td>
<td>ESP-Protocols</td>
<td>2163</td>
</tr>
<tr>
<td>7.2.9</td>
<td>ESP-BSP</td>
<td>2164</td>
</tr>
<tr>
<td>7.2.10</td>
<td>ESP-IDF-CXX</td>
<td>2164</td>
</tr>
<tr>
<td>8</td>
<td>Contributions Guide</td>
<td>2165</td>
</tr>
<tr>
<td>8.1</td>
<td>How to Contribute</td>
<td>2165</td>
</tr>
<tr>
<td>8.2</td>
<td>Before Contributing</td>
<td>2165</td>
</tr>
<tr>
<td>8.3</td>
<td>Pull Request Process</td>
<td>2165</td>
</tr>
<tr>
<td>8.4</td>
<td>Legal Part</td>
<td>2166</td>
</tr>
<tr>
<td>8.5</td>
<td>Related Documents</td>
<td>2166</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Espressif IoT Development Framework Style Guide</td>
<td>2166</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Install Pre-commit Hook for ESP-IDF Project</td>
<td>2174</td>
</tr>
<tr>
<td>8.5.3</td>
<td>Documenting Code</td>
<td>2175</td>
</tr>
<tr>
<td>8.5.4</td>
<td>Creating Examples</td>
<td>2180</td>
</tr>
<tr>
<td>8.5.5</td>
<td>API Documentation Template</td>
<td>2181</td>
</tr>
<tr>
<td>8.5.6</td>
<td>Contributor Agreement</td>
<td>2183</td>
</tr>
<tr>
<td>8.5.7</td>
<td>Copyright Header Guide</td>
<td>2185</td>
</tr>
<tr>
<td>8.5.8</td>
<td>pytest in ESP-IDF</td>
<td>2187</td>
</tr>
<tr>
<td>9</td>
<td>ESP-IDF Versions</td>
<td>2199</td>
</tr>
</tbody>
</table>
This is the documentation for Espressif IoT Development Framework (esp-idf). ESP-IDF is the official development framework for the ESP32, ESP32-S, ESP32-C, ESP32-H and ESP32-P Series SoCs.

This document describes using ESP-IDF with the ESP32-S2 SoC.

<table>
<thead>
<tr>
<th>Get Started</th>
<th>API Reference</th>
<th>API Guides</th>
</tr>
</thead>
</table>

Table of contents
Chapter 1

Get Started

This document is intended to help you set up the software development environment for the hardware based on the ESP32-S2 chip by Espressif. After that, a simple example will show you how to use ESP-IDF (Espressif IoT Development Framework) for menu configuration, then for building and flashing firmware onto an ESP32-S2 board.

Note: This is documentation for the master branch (latest version) of ESP-IDF. This version is under continual development. Stable version documentation is available, as well as other ESP-IDF Versions.

1.1 Introduction

ESP32-S2 is a system on a chip that integrates the following features:

- Wi-Fi (2.4 GHz band)
- High performance single core Xtensa® 32-bit LX7 CPU
- Ultra Low Power co-processor running either RISC-V or FSM core
- Multiple peripherals
- Built-in security hardware
- USB OTG interface

Powered by 40 nm technology, ESP32-S2 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-S2 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

1.2.1 Hardware

- An ESP32-S2 board.
- USB cable - USB A / micro USB B.
- Computer running Windows, Linux, or macOS.

Note: Currently, some of the development boards are using USB Type C connectors. Be sure you have the correct cable to connect your board!
If you have one of ESP32-S2 official development boards listed below, you can click on the link to learn more about the hardware.

**ESP32-S2-Saola-1**

This user guide provides information on ESP32-S2-Saola-1, a small-sized ESP32-S2 Datasheet based development board produced by Espressif.

![ESP32-S2-Saola-1](image)

**Fig. 1: ESP32-S2-Saola-1**

The document consists of the following major sections:

- **Getting started**: Provides an overview of the ESP32-S2-Saola-1 and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP32-S2-Saola-1’s hardware.
- **Hardware Revision Details**: Revision history, known issues, and links to user guides for previous versions (if any) of ESP32-S2-Saola-1.
- **Related Documents**: Gives links to related documentation.

**Getting Started**  This section describes how to get started with ESP32-S2-Saola-1. It begins with a few introductory sections about the ESP32-S2-Saola-1, then Section *Start Application Development* provides instructions on how to get the ESP32-S2-Saola-1 ready and flash firmware into it.

**Overview**  ESP32-S2-Saola-1 is a small-sized ESP32-S2 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-S2-Saola-1 on a breadboard.

To cover a wide range of users’ needs, ESP32-S2-Saola-1 supports:

- ESP32-S2-WROVER
- ESP32-S2-WROVER-I
- ESP32-S2-WROOM
- ESP32-S2-WROOM-I

In this guide, we take ESP32-S2-Saola-1 equipped with ESP32-S2-WROVER as an example.

**Contents and Packaging**
Retail Orders  If you order a few samples, each ESP32-S2-Saola-1 comes in an individual package in either anti-static bag or any packaging depending on your retailer. For retail orders, please go to https://www.espressif.com/en/contact-us/get-samples.

Wholesale Orders  If you order in bulk, the boards come in large cardboard boxes. For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

Description of Components  The key components of the board are described in a clockwise direction.

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-S2-WROVER</td>
<td>ESP32-S2-WROVER is a powerful, generic Wi-Fi MCU module that integrates ESP32-S2. It has a PCB antenna, a 4 MB external SPI flash and an additional 2 MB PSRAM.</td>
</tr>
<tr>
<td>Pin Headers</td>
<td>All available GPIO pins (except for the SPI bus for flash and PSRAM) are broken out to the pin headers on the board. Users can program ESP32-S2 chip to enable multiple functions such as SPI, I2S, UART, I2C, touch sensors, PWM etc.</td>
</tr>
<tr>
<td>3.3 V Power On LED</td>
<td>Turns on when the USB power is connected to the board.</td>
</tr>
<tr>
<td>USB-to-UART Bridge</td>
<td>Single USB-UART bridge chip provides transfer rates up to 3 Mbps.</td>
</tr>
<tr>
<td>Reset Button</td>
<td>Reset button.</td>
</tr>
<tr>
<td>Micro-USB Port</td>
<td>USB interface. Power supply for the board as well as the communication interface between a computer and the ESP32-S2 chip.</td>
</tr>
<tr>
<td>Boot Button</td>
<td>Download button. Holding down Boot and then pressing Reset initiates Firmware Download mode for downloading firmware through the serial port.</td>
</tr>
<tr>
<td>RGB LED</td>
<td>Addressable RGB LED (WS2812), driven by GPIO18.</td>
</tr>
</tbody>
</table>

Start Application Development  Before powering up your ESP32-S2-Saola-1, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- ESP32-S2-Saola-1
- USB 2.0 cable (Standard-A to Micro-B)
- Computer running Windows, Linux, or macOS
**Note:** Be sure to use an appropriate USB cable. Some cables are for charging only and do not provide the needed data lines nor work for programming the boards.

**Software Setup** Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment and then flash an application example into your ESP32-S2-Saola-1.

**Note:** ESP32-S2 series of chips only supports ESP-IDF master or version v4.2 and higher.

**Hardware Reference**

**Block Diagram** A block diagram below shows the components of ESP32-S2-Saola-1 and their interconnections.

![Block Diagram of ESP32-S2-Saola-1](image)

Fig. 3: ESP32-S2-Saola-1 (click to enlarge)

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

- Micro-USB port, default power supply
- 5V and GND pin headers
- 3V3 and GND pin headers

It is recommended to use the first option: Micro-USB Port.

**Header Block** The two tables below provide the Name and Function of the pin headers on both sides of the board (J2 and J3). The pin header names are shown in *ESP32-S2-Saola-1 - front*. The numbering is the same as in the ESP32-S2-Saola-1 Schematics (PDF).
### J2

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3V3</td>
<td>P</td>
<td>3.3 V power supply</td>
</tr>
<tr>
<td>2</td>
<td>IO0</td>
<td>I/O</td>
<td>GPIO0, Boot</td>
</tr>
<tr>
<td>3</td>
<td>IO1</td>
<td>I/O</td>
<td>GPIO1, ADC1_CH0, TOUCH_CH1</td>
</tr>
<tr>
<td>4</td>
<td>IO2</td>
<td>I/O</td>
<td>GPIO2, ADC1_CH1, TOUCH_CH2</td>
</tr>
<tr>
<td>5</td>
<td>IO3</td>
<td>I/O</td>
<td>GPIO3, ADC1_CH2, TOUCH_CH3</td>
</tr>
<tr>
<td>6</td>
<td>IO4</td>
<td>I/O</td>
<td>GPIO4, ADC1_CH3, TOUCH_CH4</td>
</tr>
<tr>
<td>7</td>
<td>IO5</td>
<td>I/O</td>
<td>GPIO5, ADC1_CH4, TOUCH_CH5</td>
</tr>
<tr>
<td>8</td>
<td>IO6</td>
<td>I/O</td>
<td>GPIO6, ADC1_CH5, TOUCH_CH6</td>
</tr>
<tr>
<td>9</td>
<td>IO7</td>
<td>I/O</td>
<td>GPIO7, ADC1_CH6, TOUCH_CH7</td>
</tr>
<tr>
<td>10</td>
<td>IO8</td>
<td>I/O</td>
<td>GPIO8, ADC1_CH7, TOUCH_CH8</td>
</tr>
<tr>
<td>11</td>
<td>IO9</td>
<td>I/O</td>
<td>GPIO9, ADC1_CH8, TOUCH_CH9</td>
</tr>
<tr>
<td>12</td>
<td>IO10</td>
<td>I/O</td>
<td>GPIO10, ADC1_CH9, TOUCH_CH10</td>
</tr>
<tr>
<td>13</td>
<td>IO11</td>
<td>I/O</td>
<td>GPIO11, ADC2_CH0, TOUCH_CH11</td>
</tr>
<tr>
<td>14</td>
<td>IO12</td>
<td>I/O</td>
<td>GPIO12, ADC2_CH1, TOUCH_CH12</td>
</tr>
<tr>
<td>15</td>
<td>IO13</td>
<td>I/O</td>
<td>GPIO13, ADC2_CH2, TOUCH_CH13</td>
</tr>
<tr>
<td>16</td>
<td>IO14</td>
<td>I/O</td>
<td>GPIO14, ADC2_CH3, TOUCH_CH14</td>
</tr>
<tr>
<td>17</td>
<td>IO15</td>
<td>I/O</td>
<td>GPIO15, ADC2_CH4, XTAL_32K_P</td>
</tr>
<tr>
<td>18</td>
<td>IO16</td>
<td>I/O</td>
<td>GPIO16, ADC2_CH5, XTAL_32K_N</td>
</tr>
<tr>
<td>19</td>
<td>IO17</td>
<td>I/O</td>
<td>GPIO17, ADC2_CH6, DAC_1</td>
</tr>
<tr>
<td>20</td>
<td>5V0</td>
<td>P</td>
<td>5 V power supply</td>
</tr>
<tr>
<td>21</td>
<td>GND</td>
<td>G</td>
<td>Ground</td>
</tr>
</tbody>
</table>

### J3

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GND</td>
<td>G</td>
<td>Ground</td>
</tr>
<tr>
<td>2</td>
<td>RST</td>
<td>I</td>
<td>CHIP_PU, Reset</td>
</tr>
<tr>
<td>3</td>
<td>IO46</td>
<td>I</td>
<td>GPIO46</td>
</tr>
<tr>
<td>4</td>
<td>IO45</td>
<td>I/O</td>
<td>GPIO45</td>
</tr>
<tr>
<td>5</td>
<td>IO44</td>
<td>I/O</td>
<td>GPIO44, U0RXD</td>
</tr>
<tr>
<td>6</td>
<td>IO43</td>
<td>I/O</td>
<td>GPIO43, U0TXD</td>
</tr>
<tr>
<td>7</td>
<td>IO42</td>
<td>I/O</td>
<td>GPIO42, MTMS</td>
</tr>
<tr>
<td>8</td>
<td>IO41</td>
<td>I/O</td>
<td>GPIO41, MTDI</td>
</tr>
<tr>
<td>9</td>
<td>IO40</td>
<td>I/O</td>
<td>GPIO40, MTDO</td>
</tr>
<tr>
<td>10</td>
<td>IO39</td>
<td>I/O</td>
<td>GPIO39, MTCK</td>
</tr>
<tr>
<td>11</td>
<td>IO38</td>
<td>I/O</td>
<td>GPIO38</td>
</tr>
<tr>
<td>12</td>
<td>IO37</td>
<td>I/O</td>
<td>GPIO37</td>
</tr>
<tr>
<td>13</td>
<td>IO36</td>
<td>I/O</td>
<td>GPIO36</td>
</tr>
<tr>
<td>14</td>
<td>IO35</td>
<td>I/O</td>
<td>GPIO35</td>
</tr>
<tr>
<td>16</td>
<td>IO34</td>
<td>I/O</td>
<td>GPIO34</td>
</tr>
<tr>
<td>17</td>
<td>IO33</td>
<td>I/O</td>
<td>GPIO33</td>
</tr>
<tr>
<td>17</td>
<td>IO26</td>
<td>I/O</td>
<td>GPIO26</td>
</tr>
<tr>
<td>18</td>
<td>IO21</td>
<td>I/O</td>
<td>GPIO21</td>
</tr>
<tr>
<td>19</td>
<td>IO20</td>
<td>I/O</td>
<td>GPIO20, ADC2_CH9, USB_D+</td>
</tr>
<tr>
<td>20</td>
<td>IO19</td>
<td>I/O</td>
<td>GPIO19, ADC2_CH8, USB_D-</td>
</tr>
<tr>
<td>21</td>
<td>IO18</td>
<td>I/O</td>
<td>GPIO18, ADC2_CH7, DAC_2, RGB LED</td>
</tr>
</tbody>
</table>

**Pin Layout**

**Hardware Revision Details**  This is the first revision of this board released.

---

1 P: Power supply; I: Input; O: Output; T: High impedance.
This user guide provides information on Espressif’s small-sized development board ESP32-S2-DevKitM-1. ESP32-S2-DevKitM-1 is entry-level development board. Most of the I/O pins on the module are broken out to the pin headers on both sides for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-S2-DevKitM-1 on a breadboard.
Chapter 1. Get Started

The document consists of the following major sections:

- **Getting started**: Provides an overview of the ESP32-S2-DevKitM-1 and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP32-S2-DevKitM-1’s hardware.
- **Hardware Revision Details**: Revision history, known issues, and links to user guides for previous versions (if any) of ESP32-S2-DevKitM-1.
- **Related Documents**: Gives links to related documentation.

**Getting Started** This section describes how to get started with ESP32-S2-DevKitM-1. It begins with a few introductory sections about the ESP32-S2-DevKitM-1, then Section *Start Application Development* provides instructions on how to get the ESP32-S2-DevKitM-1 ready and flash firmware into it.

**Contents and Packaging**

**Ordering Information** The development board has a variety of variants to choose from, as shown in the table below.

<table>
<thead>
<tr>
<th>Ordering Code</th>
<th>On-board Module1</th>
<th>Flash</th>
<th>PSRAM</th>
<th>Antenna</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-S2-DevKitM-1-N4R2</td>
<td>ESP32-S2-MINI-2</td>
<td>4 MB</td>
<td>2 MB</td>
<td>PCB on-board antenna</td>
</tr>
<tr>
<td>ESP32-S2-DevKitM-1U-N4R2</td>
<td>ESP32-S2-MINI-2U</td>
<td>4 MB</td>
<td>2 MB</td>
<td>External antenna connector</td>
</tr>
<tr>
<td>ESP32-S2-DevKitM-1</td>
<td>ESP32-S2-MINI-1</td>
<td>4 MB</td>
<td>—</td>
<td>PCB on-board antenna</td>
</tr>
<tr>
<td>ESP32-S2-DevKitM-1U</td>
<td>ESP32-S2-MINI-1U</td>
<td>4 MB</td>
<td>—</td>
<td>External antenna connector</td>
</tr>
<tr>
<td>ESP32-S2-DevKitM-1R</td>
<td>ESP32-S2-MINI-1</td>
<td>4 MB</td>
<td>2 MB</td>
<td>PCB on-board antenna</td>
</tr>
<tr>
<td>ESP32-S2-DevKitM-1RU</td>
<td>ESP32-S2-MINI-1U</td>
<td>4 MB</td>
<td>2 MB</td>
<td>External antenna connector</td>
</tr>
</tbody>
</table>

**Retail Orders** If you order a few samples, each ESP32-S2-DevKitM-1 comes in an individual package in either antistatic bag or any packaging depending on your retailer.


1 The ESP32-S2-MINI-2 and ESP32-S2-MINI-2U modules use chip revision v1.0, and the rest use chip revision v0.0. For more information about chip revisions, please refer to ESP32-S2 Series SoC Errata.
Wholesale Orders  If you order in bulk, the boards come in large cardboard boxes. For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

**Description of Components**  The key components of the board are described in a clockwise direction.
Chapter 1. Get Started

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-board module (ESP32-S2-MINI-1 or ESP32-S2-MINI-1U in figures above)</td>
<td>ESP32-S2-MINI series modules with an on-board PCB antenna or a connector for an external antenna. This series of modules, known for its small size, have a flash and/or a PSRAM integrated in the chip package. For more information, please refer to Ordering Information.</td>
</tr>
<tr>
<td>Pin Headers</td>
<td>All available GPIO pins (except for the SPI bus for flash) are broken out to the pin headers on the board. Users can program ESP32-S2FH4 chip to enable multiple functions such as SPI, I2S, UART, I2C, touch sensors, PWM etc. For details, please see Header Block.</td>
</tr>
<tr>
<td>3.3 V Power On LED</td>
<td>Turns on when the USB power is connected to the board.</td>
</tr>
<tr>
<td>USB to UART Bridge</td>
<td>Single USB-UART bridge chip provides transfer rates up to 3 Mbps.</td>
</tr>
<tr>
<td>Reset Button</td>
<td>Reset button.</td>
</tr>
<tr>
<td>Micro-USB Port</td>
<td>USB interface. Power supply for the board as well as the communication interface between a computer and the ESP32-S2FH4 chip.</td>
</tr>
<tr>
<td>Boot Button</td>
<td>Download button. Holding down Boot and then pressing Reset initiates Firmware Download mode for downloading firmware through the serial port.</td>
</tr>
<tr>
<td>RGB LED</td>
<td>Addressable RGB LED, driven by GPIO18.</td>
</tr>
<tr>
<td>5 V to 3.3 V LDO</td>
<td>Power regulator that converts a 5 V supply into a 3.3 V output.</td>
</tr>
<tr>
<td>External Antenna Connector</td>
<td>On ESP32-S2-MINI-2U and ESP32-S2-MINI-1U module only. For connector dimensions, please refer to Section External Antenna Connector Dimensions in module datasheet.</td>
</tr>
</tbody>
</table>

Start Application Development Before powering up your ESP32-S2-DevKitM-1, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- ESP32-S2-DevKitM-1
- USB 2.0 cable (Standard-A to Micro-B)
- Computer running Windows, Linux, or macOS

**Note:** Be sure to use an appropriate USB cable. Some cables are for charging only and do not provide the needed data lines nor work for programming the boards.

Software Setup Please proceed to Get Started, where Section Installation will quickly help you set up the development environment and then flash an application example into your ESP32-S2-DevKitM-1.

**Note:** ESP32-S2 series of chips only is only supported in ESP-IDF master or version v4.2 and higher.

Hardware Reference

Block Diagram A block diagram below shows the components of ESP32-S2-DevKitM-1 and their interconnections.

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

- Micro-USB Port, default power supply
- 5V and GND pin headers
- 3V3 and GND pin headers

It is recommended to use the first option: micro USB Port.
The two tables below provide the **Name** and **Function** of the pin headers on both sides of the board (J1 and J3). The pin header names are shown in *ESP32-S2-DevKitM-1 - front*. The numbering is the same as in the *ESP32-S2-DevKitM-1 Schematics* (PDF).

### J1

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3V3</td>
<td>P</td>
<td>3.3 V power supply</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>I/O/T</td>
<td>RTC_GPIO0, GPIO0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>I/O/T</td>
<td>RTC_GPIO1, GPIO1, TOUCH1, ADC1_CH0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>I/O/T</td>
<td>RTC_GPIO2, GPIO2, TOUCH2, ADC1_CH1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>I/O/T</td>
<td>RTC_GPIO3, GPIO3, TOUCH3, ADC1_CH2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>I/O/T</td>
<td>RTC_GPIO4, GPIO4, TOUCH4, ADC1_CH3</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>I/O/T</td>
<td>RTC_GPIO5, GPIO5, TOUCH5, ADC1_CH4</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>I/O/T</td>
<td>RTC_GPIO6, GPIO6, TOUCH6, ADC1_CH5</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>I/O/T</td>
<td>RTC_GPIO7, GPIO7, TOUCH7, ADC1_CH6</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>I/O/T</td>
<td>RTC_GPIO8, GPIO8, TOUCH8, ADC1_CH7</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>I/O/T</td>
<td>RTC_GPIO9, GPIO9, TOUCH9, ADC1_CH8, FSPIC0, FSPIC1, FSPIO4</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>I/O/T</td>
<td>RTC_GPIO10, GPIO10, TOUCH10, ADC1_CH9, FSPIC0, FSPIO4</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>I/O/T</td>
<td>RTC_GPIO11, GPIO11, TOUCH11, ADC2_CH0, FSPID, FSPIC1, FSPIO4</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>I/O/T</td>
<td>RTC_GPIO12, GPIO12, TOUCH12, ADC2_CH1, FSPICLKG, FSPIO4</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>I/O/T</td>
<td>RTC_GPIO13, GPIO13, TOUCH13, ADC2_CH2, FSPIODGQ</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>I/O/T</td>
<td>RTC_GPIO14, GPIO14, TOUCH14, ADC2_CH3, FSPICLKG, FSPIO4</td>
</tr>
<tr>
<td>17</td>
<td>15</td>
<td>I/O/T</td>
<td>RTC_GPIO15, GPIO15, U0RTS, ADC2_CH4, XTAL_32K_P</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>I/O/T</td>
<td>RTC_GPIO16, GPIO16, U0CTS, ADC2_CH5, XTAL_32K_N</td>
</tr>
<tr>
<td>19</td>
<td>17</td>
<td>I/O/T</td>
<td>RTC_GPIO17, GPIO17, U1TXD, ADC2_CH6, DAC_1</td>
</tr>
<tr>
<td>20</td>
<td>5V</td>
<td>P</td>
<td>5 V power supply</td>
</tr>
<tr>
<td>21</td>
<td>G</td>
<td>G</td>
<td>Ground</td>
</tr>
</tbody>
</table>

2 P: Power supply; I: Input; O: Output; T: High impedance.
## Chapter 1. Get Started

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G</td>
<td>G</td>
<td>Ground</td>
</tr>
<tr>
<td>2</td>
<td>RST</td>
<td>I</td>
<td>CHIP_PU</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>I</td>
<td>GPIO46</td>
</tr>
<tr>
<td>4</td>
<td>45</td>
<td>I/O/T</td>
<td>GPIO45</td>
</tr>
<tr>
<td>5</td>
<td>RX</td>
<td>I/O/T</td>
<td>U0RXD, GPIO44, CLK_OUT2</td>
</tr>
<tr>
<td>6</td>
<td>TX</td>
<td>I/O/T</td>
<td>U0TXD, GPIO43, CLK_OUT1</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>I/O/T</td>
<td>MTMS, GPIO42</td>
</tr>
<tr>
<td>8</td>
<td>41</td>
<td>I/O/T</td>
<td>MTDI, GPIO41, CLK_OUT1</td>
</tr>
<tr>
<td>9</td>
<td>40</td>
<td>I/O/T</td>
<td>MTD0, GPIO40, CLK_OUT2</td>
</tr>
<tr>
<td>10</td>
<td>39</td>
<td>I/O/T</td>
<td>MTCK, GPIO39, CLK_OUT3</td>
</tr>
<tr>
<td>11</td>
<td>38</td>
<td>I/O/T</td>
<td>GPIO38, FSPIWP</td>
</tr>
<tr>
<td>12</td>
<td>37</td>
<td>I/O/T</td>
<td>SPI0QS, GPIO37, FSPIQ</td>
</tr>
<tr>
<td>13</td>
<td>36</td>
<td>I/O/T</td>
<td>SPI07, GPIO36, FSPICLK</td>
</tr>
<tr>
<td>14</td>
<td>35</td>
<td>I/O/T</td>
<td>SPI06, GPIO35, FSPID</td>
</tr>
<tr>
<td>15</td>
<td>34</td>
<td>I/O/T</td>
<td>SPI05, GPIO34, FSPICS0</td>
</tr>
<tr>
<td>16</td>
<td>33</td>
<td>I/O/T</td>
<td>SPI04, GPIO33, FSPIHD</td>
</tr>
<tr>
<td>17</td>
<td>26</td>
<td>I/O/T</td>
<td>SPI01, GPIO26</td>
</tr>
<tr>
<td>18</td>
<td>21</td>
<td>I/O/T</td>
<td>RTC_GPIO21, GPIO21</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>I/O/T</td>
<td>RTC_GPIO20, GPIO20, U1CTS, ADC2_CH9, CLK_OUT1, USB_D+</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>I/O/T</td>
<td>RTC_GPIO19, GPIO19, U1RTS, ADC2_CH8, CLK_OUT2, USB_D-</td>
</tr>
<tr>
<td>21</td>
<td>18</td>
<td>I/O/T</td>
<td>RTC_GPIO18, GPIO18, U1RXD, ADC2_CH7, DAC_2, CLK_OUT3, RGB LED</td>
</tr>
</tbody>
</table>

**Fig. 8: ESP32-S2-DevKitM-1 Pin Layout (click to enlarge)**

---

**ESP32-S2 DevKitM-1**

**ESP32-S2 Specs**
- 32-bit Xtensa single-core @240MHz
- Wi-Fi IEEE 802.11 b/g/n 2.4GHz
- 32KB SRAM (16 KB SRAM in RTC)
- 128 KB ROM
- 43 GPIOs, 4x SPI, 2x UART, 2x I2C, Touch, I2S, RMT, LED PWM, USB-OTG,
- TWAI, 2x 8-bit DAC, 12-bit ADC

**Pin Layout**

- PMW Capable Pin
- GPIO Input Only
- GPIO Input and Output
- UART for Debugging and USB
- ADC 0 Analog-to-Digital Converter
- ADC 1 Other Related Functions
- ADC 2 Serial for Debug/Programming
- DAC 0 Digital-to-Analog Converter
- DAC 1 Stripping Pin Functions
- RTC, Power Domain (S0/S0P3_RST)
- GND Ground
- MISO Clock Output
- VCC Power Rails (3V3 and 5V)
Chapter 1. Get Started

**Hardware Revision Details**  This is the first revision of this board released.

**Related Documents**

- ESP32-S2 Series Chip Revision v1.0 Datasheet (PDF)
- ESP32-S2 Series Chip Revision v0.0 Datasheet (PDF)
- ESP32-S2 Series SoC Errata (PDF)
- ESP32-S2-MINI-2 & ESP32-S2-MINI-2U Module Datasheet (PDF)
- ESP32-S2-MINI-1 & ESP32-S2-MINI-1U Module Datasheet (PDF)
- ESP32-S2-DevKitM-1 Schematics (PDF)
- ESP32-S2-DevKitM-1 PCB Layout (PDF)
- ESP32-S2-DevKitM-1 Dimensions (PDF)
- ESP Product Selector

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

**ESP32-S2-DevKitC-1**

This user guide will help you get started with ESP32-S2-DevKitC-1 and will also provide more in-depth information. ESP32-S2-DevKitC-1 is an entry-level development board. This board integrates complete Wi-Fi functions. 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-S2-DevKitC-1 on a breadboard.

![Fig. 9: ESP32-S2-DevKitC-1 with the ESP32-S2-SOLO Module](image)

The document consists of the following major sections:

- **Getting Started**: Overview of ESP32-S2-DevKitC-1 and hardware/software setup instructions to get started.
- **Hardware Reference**: More detailed information about the ESP32-S2-DevKitC-1’s hardware.
- **Hardware Revision Details**: Revision history, known issues, and links to user guides for previous versions (if any) of ESP32-S2-DevKitC-1.
- **Related Documents**: Links to related documentation.
Getting Started  
This section provides a brief introduction of ESP32-S2-DevKitC-1, instructions on how to do the initial hardware setup and how to flash firmware onto it.

![ESP32-S2-DevKitC-1-front](image)

Description of Components  
The key components of the board are described in a clockwise direction.

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-board module (ESP32-S2-SOLO or ESP32-S2-SOLO-U in the above figure)</td>
<td>ESP32-S2-SOLO series modules with an on-board PCB antenna or a connector for an external antenna. This series of modules have multiple options for flash and PSRAM size. For more information, please refer to Ordering Information.</td>
</tr>
<tr>
<td>3.3 V Power On LED</td>
<td>Turns on when the USB power is connected to the board.</td>
</tr>
<tr>
<td>USB-to-UART Bridge</td>
<td>Single USB-to-UART bridge chip provides transfer rates up to 3 Mbps.</td>
</tr>
<tr>
<td>Pin Headers</td>
<td>All available GPIO pins (except for the SPI bus for flash) are broken out to the pin headers on the board. For details, please see Header Block.</td>
</tr>
<tr>
<td>ESP32-S2 USB Port</td>
<td>ESP32-S2 full-speed USB OTG interface, compliant with the USB 1.1 specifications. The interface is used for power supply to the board, for flashing applications to the chip, and for communication with the chip using USB 1.1 protocols.</td>
</tr>
<tr>
<td>Reset Button</td>
<td>Press this button to restart the system.</td>
</tr>
<tr>
<td>Boot Button</td>
<td>Download button.  Holding down Boot and then pressing Reset initiates Firmware Download mode for downloading firmware through the serial port.</td>
</tr>
<tr>
<td>USB-to-UART Port</td>
<td>A Micro-USB port used for power supply to the board, for flashing applications to the chip, as well as the communication with the chip via the on-board USB-to-UART bridge.</td>
</tr>
<tr>
<td>RGB LED</td>
<td>Addressable RGB LED, driven by GPIO18.</td>
</tr>
<tr>
<td>5 V to 3.3 V LDO</td>
<td>Power regulator that converts a 5 V supply into a 3.3 V output.</td>
</tr>
</tbody>
</table>

Start Application Development  
Before powering up your ESP32-S2-DevKitC-1, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- ESP32-S2-DevKitC-1
Chapter 1. Get Started

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

**Note:** Be sure to use an appropriate USB cable. Some cables are for charging only and do not provide the needed data lines nor work for programming the boards.

**Hardware Setup** Connect the board with the computer using **USB-to-UART Port** or **ESP32-S2 USB Port**. In subsequent steps, **USB-to-UART Port** will be used by default.

**Software Setup** Please proceed to **ESP-IDF Get Started**, where Section **Installation Step by Step** will quickly help you set up the development environment and then flash an application example into your ESP32-S2-DevKitC-1.

**Contents and Packaging**

**Ordering Information** The development board has a variety of variants to choose from, as shown in the table below.

<table>
<thead>
<tr>
<th>Ordering Code</th>
<th>On-board Module¹</th>
<th>Flash</th>
<th>PSRAM</th>
<th>Antenna</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-S2-DevKitC-1-N8R2</td>
<td>ESP32-S2-SOLO-2</td>
<td>8 MB</td>
<td>2 MB</td>
<td>PCB on-board antenna</td>
</tr>
<tr>
<td></td>
<td>(Recommended)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP32-S2-DevKitC-1U-N8R2</td>
<td>ESP32-S2-SOLO-2U</td>
<td>8 MB</td>
<td>2 MB</td>
<td>External antenna connector</td>
</tr>
<tr>
<td></td>
<td>(Recommended)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP32-S2-DevKitC-1</td>
<td>ESP32-S2-SOLO</td>
<td>4 MB</td>
<td>—</td>
<td>PCB on-board antenna</td>
</tr>
<tr>
<td></td>
<td>(Recommended)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP32-S2-DevKitC-1U</td>
<td>ESP32-S2-SOLO-U</td>
<td>4 MB</td>
<td>—</td>
<td>External antenna connector</td>
</tr>
<tr>
<td></td>
<td>(Recommended)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP32-S2-DevKitC-1R</td>
<td>ESP32-S2-SOLO</td>
<td>4 MB</td>
<td>2 MB</td>
<td>PCB on-board antenna</td>
</tr>
<tr>
<td></td>
<td>(Recommended)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESP32-S2-DevKitC-1RU</td>
<td>ESP32-S2-SOLO-U</td>
<td>4 MB</td>
<td>2 MB</td>
<td>External antenna connector</td>
</tr>
</tbody>
</table>

**Retail Orders** If you order a few samples, each ESP32-S2-DevKitC-1 comes in an individual package in either antistatic bag or any packaging depending on your retailer.


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

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

**Hardware Reference**

**Block Diagram** The block diagram below shows the components of ESP32-S2-DevKitC-1 and their interconnections.

¹ The ESP32-S2-SOLO-2 and ESP32-S2-SOLO-2U modules use chip revision v1.0, and the rest use chip revision v0.0. For more information about chip revisions, please refer to ESP32-S2 Series SoC Errata.
### Power Supply Options

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

- USB-to-UART Port and ESP32-S2 USB Port (either one or both), default power supply (recommended)
- 5V and G (GND) pins
- 3.3V and G (GND) pins

### Header Block

The two tables below provide the **Name** and **Function** of the pin headers on both sides of the board (J1 and J3). The pin header names are shown in *ESP32-S2-DevKitC-1 - front*. The numbering is the same as in the *ESP32-S2-DevKitC-1 Schematic* (PDF).

#### J1

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3V3</td>
<td>P</td>
<td>3.3 V power supply</td>
</tr>
<tr>
<td>2</td>
<td>3V3</td>
<td>P</td>
<td>3.3 V power supply</td>
</tr>
<tr>
<td>3</td>
<td>RST</td>
<td>I</td>
<td>CHIP_PU</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>I/O/T</td>
<td>RTC_GPIO4, GPIO4, TOUCH4, ADC1_CH3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>I/O/T</td>
<td>RTC_GPIO5, GPIO5, TOUCH5, ADC1_CH4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>I/O/T</td>
<td>RTC_GPIO6, GPIO6, TOUCH6, ADC1_CH5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>I/O/T</td>
<td>RTC_GPIO7, GPIO7, TOUCH7, ADC1_CH6</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>I/O/T</td>
<td>RTC_GPIO15, GPIO15, U0RTS, ADC2_CH4, XTAL_32K_P</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>I/O/T</td>
<td>RTC_GPIO16, GPIO16, U0CTS, ADC2_CH5, XTAL_32K_N</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>I/O/T</td>
<td>RTC_GPIO17, U1TXD, ADC2_CH6, DAC_1</td>
</tr>
<tr>
<td>11</td>
<td>18*</td>
<td>I/O/T</td>
<td>RTC_GPIO18, GPIO18, U1RXD, ADC2_CH7, DAC_2, CLK_OUT3, RGB LED</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>I/O/T</td>
<td>RTC_GPIO8, GPIO8, TOUCH8, ADC1_CH7</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>I/O/T</td>
<td>RTC_GPIO3, GPIO3, TOUCH3, ADC1_CH2</td>
</tr>
<tr>
<td>14</td>
<td>46</td>
<td>I</td>
<td>GPIO46</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>I/O/T</td>
<td>RTC_GPIO9, GPIO9, TOUCH9, ADC1_CH8, FSPIH D</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>I/O/T</td>
<td>RTC_GPIO10, GPIO10, TOUCH10, ADC1_CH9, FSPICS0, FSPIO4</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>I/O/T</td>
<td>RTC_GPIO11, GPIO11, TOUCH11, ADC2_CH0, FSPID, FSPIO5</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>I/O/T</td>
<td>RTC_GPIO12, GPIO12, TOUCH12, ADC2_CH1, FSPICLK, FSPIO6</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>I/O/T</td>
<td>RTC_GPIO13, GPIO13, TOUCH13, ADC2_CH2, FSPIQ, FSPIO7</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>I/O/T</td>
<td>RTC_GPIO14, GPIO14, TOUCH14, ADC2_CH3, FSPITWP, FSPIQDQS</td>
</tr>
<tr>
<td>21</td>
<td>5V</td>
<td>P</td>
<td>5 V power supply</td>
</tr>
<tr>
<td>22</td>
<td>G</td>
<td>G</td>
<td>Ground</td>
</tr>
</tbody>
</table>

2 P: Power supply; I: Input; O: Output; T: High impedance.
3 GPIO18 is not pulled up on boards with an ESP32-S2-SOLO-2 or ESP32-S2-SOLO-2U module.
### J3

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G</td>
<td>G</td>
<td>Ground</td>
</tr>
<tr>
<td>2</td>
<td>TX</td>
<td>I/O/T</td>
<td>U0TXD, GPIO43, CLK_OUT1</td>
</tr>
<tr>
<td>3</td>
<td>RX</td>
<td>I/O/T</td>
<td>U0RXD, GPIO44, CLK_OUT2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>I/O/T</td>
<td>RTC_GPIO1, GPIO1, TOUCH1, ADC1_Ch0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>I/O/T</td>
<td>RTC_GPIO2, GPIO2, TOUCH2, ADC1_Ch1</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>I/O/T</td>
<td>MTMS, GPIO42</td>
</tr>
<tr>
<td>7</td>
<td>41</td>
<td>I/O/T</td>
<td>MTDI, GPIO41, CLK_OUT1</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>I/O/T</td>
<td>MTDO, GPIO40, CLK_OUT2</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>I/O/T</td>
<td>MTCK, GPIO39, CLK_OUT3</td>
</tr>
<tr>
<td>10</td>
<td>38</td>
<td>I/O/T</td>
<td>GPIO38, FSPIWP</td>
</tr>
<tr>
<td>11</td>
<td>37</td>
<td>I/O/T</td>
<td>SPIDQS, GPIO37, FSPIQ</td>
</tr>
<tr>
<td>12</td>
<td>36</td>
<td>I/O/T</td>
<td>SPIIO7, GPIO36, FSPICLK</td>
</tr>
<tr>
<td>13</td>
<td>35</td>
<td>I/O/T</td>
<td>SPIIO6, GPIO35, FSPID</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>I/O/T</td>
<td>RTC_GPIO0, GPIO0</td>
</tr>
<tr>
<td>15</td>
<td>45</td>
<td>I/O/T</td>
<td>GPIO45</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
<td>I/O/T</td>
<td>SPIIO5, GPIO34, FSPICS0</td>
</tr>
<tr>
<td>17</td>
<td>33</td>
<td>I/O/T</td>
<td>SPIIO4, GPIO33, FSPiHD</td>
</tr>
<tr>
<td>18</td>
<td>21</td>
<td>I/O/T</td>
<td>RTC_GPIO21, GPIO21</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>I/O/T</td>
<td>RTC_GPIO20, GPIO20, U1CTS, ADC2_Ch9, CLK_OUT1, USB_D+</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>I/O/T</td>
<td>RTC_GPIO19, GPIO19, U1RTS, ADC2_Ch8, CLK_OUT2, USB_D-</td>
</tr>
<tr>
<td>21</td>
<td>G</td>
<td>G</td>
<td>Ground</td>
</tr>
<tr>
<td>22</td>
<td>G</td>
<td>G</td>
<td>Ground</td>
</tr>
</tbody>
</table>

Fig. 12: ESP32-S2-DevKitC-1 Pin Layout (click to enlarge)
Chapter 1. Get Started

**Hardware Revision Details**  This is the first revision of this board released.

**Related Documents**

- ESP32-S2 Series Chip Revision v1.0 Datasheet (PDF)
- ESP32-S2 Series Chip Revision v0.0 Datasheet (PDF)
- ESP32-S2 Series SoC Errata (PDF)
- ESP32-S2-SOLO-2 & ESP32-S2-SOLO-2U Module Datasheet (PDF)
- ESP32-S2-SOLO & ESP32-S2-SOLO-U Module Datasheet (PDF)
- ESP32-S2-DevKitC-1 Schematic (PDF)
- ESP32-S2-DevKitC-1 PCB Layout (PDF)
- ESP32-S2-DevKitC-1 Dimensions (PDF)
- ESP32-S2-DevKitC-1 Dimensions source file (DXF) - You can view it with Autodesk Viewer online

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

**ESP32-S2-Kaluga-1 Kit v1.3**

Older version: *ESP32-S2-Kaluga-1 Kit v1.2*

The ESP32-S2-Kaluga-1 kit v1.3 is a development kit by Espressif that is mainly created to:

- Demonstrate the ESP32-S2’s human-computer interaction functionalities
- Provide the users with the tools for development of human-computer interaction applications based on the ESP32-S2

There are many ways of how the ESP32-S2’s abundant functionalities can be used. For starters, the possible use cases may include:

- **Smart home**: From simplest smart lighting, smart door locks, smart sockets, to video streaming devices, security cameras, OTT devices, and home appliances
- **Battery-powered equipment**: Wi-Fi mesh sensor networks, Wi-Fi-networked toys, wearable devices, health management equipment
- **Industrial automation equipment**: Wireless control and robot technology, intelligent lighting, HVAC control equipment, etc.
- **Retail and catering industry**: POS machines and service robots

![ESP32-S2-Kaluga-1 Kit Overview](image)

**Fig. 13: ESP32-S2-Kaluga-1-Kit Overview (click to enlarge)**

The ESP32-S2-Kaluga-1 kit consists of the following boards:
Chapter 1. Get Started

- Main board: ESP32-S2-Kaluga-1
- Extension boards:
  - ESP-LyraT-8311A v1.3 - audio player
  - ESP-LyraP-TouchA v1.1 - touch panel
  - ESP-LyraP-LCD32 v1.2 - 3.2" LCD screen
  - ESP-LyraP-CAM v1.1 - camera board

Due to the presence of multiplexed pins on ESP32-S2, certain extension board combinations have limited compatibility. For more details, please see Compatibility of Extension Boards.

This document is mostly dedicated to the main board and its interaction with the extension boards. For more detailed information on each extension board, click their respective links.

This guide covers:
- Getting Started: Provides an overview of the ESP32-S2-Kaluga-1 and hardware/software setup instructions to get started.
- Hardware reference: Provides more detailed information about the ESP32-S2-Kaluga-1’s hardware.
- Hardware Revision Details: Covers revision history, known issues, and links to user guides for previous versions of the ESP32-S2-Kaluga-1.
- Related Documents: Gives links to related documentation.

Getting Started This section describes how to get started with the ESP32-S2-Kaluga-1. It begins with a few introductory sections about the ESP32-S2-Kaluga-1, then Section Start Application Development provides instructions on how to do the initial hardware setup and then how to flash firmware onto the ESP32-S2-Kaluga-1.

Overview The ESP32-S2-Kaluga-1 main board is the heart of the kit. It integrates the ESP32-S2-WROVER module and all the connectors for extension boards. This board is the key tool in prototyping human-computer interaction interfaces.

The ESP32-S2-Kaluga-1 board has connectors for boards with:
- Extension header (ESP-LyraT-8311A, ESP-LyraP-LCD32)
- Camera header (ESP-LyraP-CAM)
- Touch FPC connector (ESP-LyraP-TouchA)
- LCD FPC connector (no official extension boards yet)
- I2C FPC connector (no official extension boards yet)

All the four extension boards are specially designed to support the following features:
- **Touch panel control**
  - Six touch buttons
  - Supports acrylic panels up to 5 mm
  - Wet hand operation
  - Water rejection, ESP32-S2 can be configured to disable all touchpads automatically if multiple pads are simultaneously covered with water and to re-enable touchpads if the water is removed
- **Audio playback**
  - Connect speakers to play audio
  - Use together with the Touch panel to control audio playback and adjust volume
- **LCD display**
  - LCD interface (8-bit parallel RGB, 8080, and 6800 interface)
- **Camera image acquisition**
  - Supports OV2640 and OV3660 camera modules
  - 8-bit DVP image sensor interface (ESP32-S2 also supports 16-bit DVP image sensors, you can design it yourself)
  - Clock frequency up to 40 MHz
  - Optimized DMA transmission bandwidth for easier transmission of high-resolution images

Description of Components The description of components starts from the ESP32-S2 module on the left side and then goes clockwise.
Fig. 14: ESP32-S2-Kaluga-1 (click to enlarge)
Fig. 15: ESP32-S2-Kaluga-1 - front (click to enlarge)

Fig. 16: ESP32-S2-Kaluga-1 - back (click to enlarge)
Reserved means that the functionality is available, but the current version of the kit does not use it.

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32-S2-WROVER Module</td>
<td>Module integrating the ESP32-S2 chip that provides Wi-Fi connectivity, data processing power, and flexible data storage.</td>
</tr>
<tr>
<td>4.3” LCD FPC Connector</td>
<td>(Reserved) Connect to a 4.3” LCD extension board using the FPC cable.</td>
</tr>
<tr>
<td>ESP Prog Connector</td>
<td>(Reserved) Connection for Espressif’s download device (ESP-Prog) to flash ESP32-S2 system.</td>
</tr>
<tr>
<td>JTAG Switch</td>
<td>Switch to ON to enable connection between ESP32-S2 and FT2232; JTAG debugging will then be possible using USB-UART/JTAG Port. See also JTAG Debugging.</td>
</tr>
<tr>
<td>Breakout Header 2</td>
<td>Some GPIO pins of the ESP32-S2-WROVER module are broken out to this header, see labels on the board.</td>
</tr>
<tr>
<td>USB-to-UART/JTAG Bridge</td>
<td>FT2232 adapter board allowing for communication over USB port using UART/JTAG protocols.</td>
</tr>
<tr>
<td>Camera Header</td>
<td>Mount a camera extension board here (e.g., ESP-LyraP-CAM).</td>
</tr>
<tr>
<td>Extension Header</td>
<td>Mount the extension boards having such connectors here.</td>
</tr>
<tr>
<td>Reset Button</td>
<td>Press this button to restart the system</td>
</tr>
<tr>
<td>Boot Button</td>
<td>Holding down Boot and then pressing Reset initiates Firmware Download mode for downloading firmware through the serial port.</td>
</tr>
<tr>
<td>USB-UART/JTAG Port</td>
<td>Communication interface (UART or JTAG) between a PC and the ESP32-S2 module.</td>
</tr>
<tr>
<td>USB Power Port</td>
<td>Power supply for the board.</td>
</tr>
<tr>
<td>Battery Port</td>
<td>Connect an external battery to the 2-pin battery connector.</td>
</tr>
<tr>
<td>Power On LED</td>
<td>Turns on when the USB or an external power supply is connected to the board.</td>
</tr>
<tr>
<td>Power Switch</td>
<td>Switch to ON to power the system.</td>
</tr>
<tr>
<td>RGB Jumper</td>
<td>To have access to the RGB LED, place a jumper onto the pins.</td>
</tr>
<tr>
<td>RGB LED</td>
<td>Programmable RGB LED and controlled by GPIO45. Before using it, you need to put RGB Jumper ON.</td>
</tr>
<tr>
<td>Power Regulator</td>
<td>Regulator converts 5 V to 3.3 V.</td>
</tr>
<tr>
<td>I2C FPC Connector</td>
<td>(Reserved) Connect to other I2C extension boards using the FPC cable.</td>
</tr>
<tr>
<td>Breakout Header 1</td>
<td>Some GPIO pins of the ESP32-S2-WROVER module are broken out to this header, see labels on the board.</td>
</tr>
<tr>
<td>Touch FPC Connector</td>
<td>Connect the ESP-LyraP-TouchA extension board using the FPC cable.</td>
</tr>
<tr>
<td>Touch Switch</td>
<td>In OFF position, GPIO1 to GPIO14 are used for connection to touch sensors; switch to ON if you want to use them for other purposes.</td>
</tr>
<tr>
<td>3.2” LCD FPC connector</td>
<td>Connect a 3.2” LCD extension board (e.g., ESP-LyraP-LCD32) using the FPC cable.</td>
</tr>
</tbody>
</table>

**Start Application Development**  Before powering up your ESP32-S2-Kaluga-1, please make sure that it is in good condition with no obvious signs of damage.

**Required Hardware**

- ESP32-S2-Kaluga-1
- Two USB 2.0 cables (Standard-A to Micro-B)
  - For power supply
  - For UART/JTAG communication
- Computer running Windows, Linux, or macOS
- Any extension boards of your choice

**Hardware Setup**

1. Connect the extension board(s) of your choice (go to their respective user guides if necessary)
2. Plug in both USB cables
3. Turn the Power Switch to ON - the Power On LED will light up
**Software Setup**  
Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment.

The programming guide and application examples for your ESP32-S2-Kaluga-1 kit can be found in *esp-dev-kits* repository on GitHub.

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

**Contents and Packaging**

**Retail Orders**  
If you order one or several samples of the kit, each ESP32-S2-Kaluga-1 development kit comes in an individual package.

![ESP32-S2-Kaluga-1 package](image)

*Fig. 17: ESP32-S2-Kaluga-1 - package*

The contents are as follows:

- **Main Board**
  - ESP32-S2-Kaluga-1

- **Extension Boards**:
  - ESP-LyraT-8311A
  - ESP-LyraP-CAM
  - ESP-LyraP-TouchA
  - ESP-LyraP-LCD32

- **Connectors**
  - 20-pin FPC cable (to connect ESP32-S2-Kaluga-1 to ESP-LyraP-TouchA)

- **Fasteners**
Chapter 1. Get Started

- Mounting bolts (x8)
- Screws (x4)
- Nuts (x4)


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

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

Hardware Reference

Block Diagram A block diagram below shows the components of the ESP32-S2-Kaluga-1 and their interconnections.

Fig. 18: ESP32-S2-Kaluga-1 block diagram

Power Supply Options There are four ways to provide power to the board:

- Micro USB port, default power supply
- External battery via the 2-pin battery connector
- 5V and GND header pins
- 3V3 and GND header pins

Compatibility of Extension Boards If you want to use more than one extension board at the same time, please check the table given below.
### Boards Used

<table>
<thead>
<tr>
<th>Boards Used</th>
<th>HW Conflict</th>
<th>Limitations</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>8311A v1.3 + CAM v1.1</td>
<td>I2S Controller</td>
<td>ESP32-S2 has only one I2S interface. But both extension boards require connection via the ESP32-S2’s I2S interface (LyraT-8311A in Standard mode, ESP-LyraP-CAM in Camera mode).</td>
<td>Utilize time division multiple access, or use a different audio module that can be connected via other GPIOs or DAC.</td>
</tr>
<tr>
<td>TouchA v1.1 + LCD32 v1.2</td>
<td>IO11, IO6</td>
<td>Touch actions cannot be triggered because of the multiplexed pin IO11. ESP-LyraP-LCD32 will not be affected because its BLCT pin will be disconnected from IO6.</td>
<td>Do not initialize IO11 (NETWORK) for your ESP-LyraP-TouchA, or configure the BLCT pin to -l (= do not use BLCT) for your ESP-LyraP-LCD32.</td>
</tr>
<tr>
<td>8311A v1.3 + LCD32 v1.2</td>
<td>IO6</td>
<td>BLCT pin of ESP32-S2-Kaluga-1 will be disconnected from IO6.</td>
<td>Configure the BK pin to -l (= do not use BLCT) for your ESP-LyraP-LCD32.</td>
</tr>
<tr>
<td>TouchA v1.1 + 8311A v1.3</td>
<td>Pin BT_ADC on ESP-LyraT-8311A</td>
<td>This pin is required for initialization of the six button on ESP-LyraT-8311A. At the same time, ESP-LyraP-TouchA needs this pin for its touch actions.</td>
<td>If you plan to use buttons on ESP-LyraT-8311A, do not initialize pin IO6 (PHOTO) for your ESP-LyraP-TouchA.</td>
</tr>
<tr>
<td>TouchA v1.1 + CAM v1.1</td>
<td>IO1, IO2, IO3</td>
<td>Cannot be used simultaneously because of the mentioned multiplexed pins.</td>
<td>For ESP-LyraP-TouchA, do not initialize IO1 (VOL_UP), IO2 (PLAY), and IO3 (VOL_DOWN).</td>
</tr>
<tr>
<td>TouchA v1.1 + LCD32 v1.2 + CAM v1.1</td>
<td>IO1, IO2, IO3, IO11</td>
<td>Conflicts on the mentioned multiplexed pins.</td>
<td>For ESP-LyraP-TouchA, do not initialize IO1 (VOL_UP), IO2 (PLAY), IO3 (VOL_DOWN), and IO11 (NETWORK).</td>
</tr>
<tr>
<td>TouchA v1.1 + LCD32 v1.2 + 8311A v1.3</td>
<td>IO6, IO11</td>
<td>If ESP-LyraT-8311A’s pin BT_ADC is used to initialize the board’s six buttons, IO6 and IO11 will not be available for the other boards.</td>
<td>Do not initialize IO11 (NETWORK) for your ESP-LyraP-TouchA. Also, if you need to use BT_ADC, do not initialize IO6 (PHOTO).</td>
</tr>
</tbody>
</table>

Also, all extension boards and the *JTAG interface* share the same pins IO39, IO40, IO41 and IO42. For this reason, the following may disturb the JTAG operation:

- Plugging in any extension board
- Debugging an application that is using an extension board

### Hardware Revision Details

#### ESP32-S2-Kaluga-1 Kit v1.3

- The following pins re-assigned to fix the download issue
  - Camera D2: GPIO36
  - Camera D3: GPIO37
  - AU_I2S1_SDI: GPIO34
  - AU_WAKE_INT: GPIO46
- RGB pin header moved to the board’s edge
- All dip switches moved to the flip side for convenient operation

#### ESP32-S2-Kaluga-1 Kit v1.2  *Initial release*

### Related Documents

---

*Espressif Systems* 26  Release master

*Submit Document Feedback*
**ESP32-S2-Kaluga-1 Kit v1.2**

New version available: *ESP32-S2-Kaluga-1 Kit v1.3*

The ESP32-S2-Kaluga-1 kit v1.2 is a development kit by Espressif that is mainly created to:

- Demonstrate the ESP32-S2’s human-computer interaction functionalities
- Provide the users with the tools for development of human-computer interaction applications based on the ESP32-S2

There are many ways of how the ESP32-S2’s abundant functionalities can be used. For starters, the possible use cases may include:

- **Smart home**: From simplest smart lighting, smart door locks, smart sockets, to video streaming devices, security cameras,OTT devices, and home appliances
- **Battery-powered equipment**: Wi-Fi mesh sensor networks, Wi-Fi-networked toys, wearable devices, health management equipment
- **Industrial automation equipment**: Wireless control and robot technology, intelligent lighting, HVAC control equipment, etc.
- **Retail and catering industry**: POS machines and service robots

![ESP32-S2-Kaluga-1-Kit Overview](fig.png)

**Fig. 19: ESP32-S2-Kaluga-1-Kit Overview (click to enlarge)**

The ESP32-S2-Kaluga-1 kit consists of the following boards:

- **Main board**: *ESP32-S2-Kaluga-1*
- **Extension boards**:
  - *ESP-LyraT-8311A v1.2* - audio player
  - *ESP-LyraP-TouchA v1.1* - touch panel
  - *ESP-LyraP-LCD32 v1.1* - 3.2” LCD screen
  - *ESP-LyraP-CAM v1.0* - camera board

Due to the presence of multiplexed pins on ESP32-S2, certain extension board combinations have limited compatibility. For more details, please see *Compatibility of Extension Boards*.

This document is mostly dedicated to the main board and its interaction with the extension boards. For more detailed information on each extension board, click their respective links.

This guide covers:

- **Getting Started**: Provides an overview of the ESP32-S2-Kaluga-1 and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP32-S2-Kaluga-1’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP32-S2-Kaluga-1.
• **Related Documents:** Gives links to related documentation.

**Getting Started** This section describes how to get started with the ESP32-S2-Kaluga-1. It begins with a few introductory sections about the ESP32-S2-Kaluga-1, then Section *Start Application Development* provides instructions on how to do the initial hardware setup and then how to flash firmware onto the ESP32-S2-Kaluga-1.

**Overview** The ESP32-S2-Kaluga-1 main board is the heart of the kit. It integrates the ESP32-S2-WROVER module and all the connectors for extension boards. This board is the key tool in prototyping human-computer interaction interfaces.

The ESP32-S2-Kaluga-1 board has connectors for boards with:

- Extension header (ESP-LyraT-8311A, ESP-LyraP-LCD32)
- Camera header (ESP-LyraP-CAM)
- Touch FPC connector (ESP-LyraP-TouchA)
- LCD FPC connector (no official extension boards yet)
- I2C FPC connector (no official extension boards yet)

![Fig. 20: ESP32-S2-Kaluga-1 (click to enlarge)](image)

All the four extension boards are specially designed to support the following features:

- **Touch panel control**
  - Six touch buttons
  - Supports acrylic panels up to 5 mm
  - Wet hand operation
  - Water rejection, ESP32-S2 can be configured to disable all touchpads automatically if multiple pads are simultaneously covered with water and to re-enable touchpads if the water is removed

- **Audio playback**
Chapter 1. Get Started

- Connect speakers to play audio
- Use together with the Touch panel to control audio playback and adjust volume

**LCD display**
- LCD interface (8-bit parallel RGB, 8080, and 6800 interface)

**Camera image acquisition**
- Supports OV2640 and OV3660 camera modules
- 8-bit DVP image sensor interface (ESP32-S2 also supports 16-bit DVP image sensors, you can design it yourself)
- Clock frequency up to 40 MHz
- Optimized DMA transmission bandwidth for easier transmission of high-resolution images

**Description of Components**
The description of components starts from the ESP32-S2 module on the left side and then goes clockwise.

**Reserved** means that the functionality is available, but the current version of the kit does not use it.
### Key Component | Description
---|---
ESP32-S2-WROVER Module | Module integrating the ESP32-S2 chip that provides Wi-Fi connectivity, data processing power, and flexible data storage.
4.3” LCD FPC Connector | (Reserved) Connect to a 4.3” LCD extension board using the FPC cable.
ESP Prog Connector | (Reserved) Connection for Espressif’s download device (ESP-Prog) to flash ESP32-S2 system.
JTAG Switch | Switch to ON to enable connection between ESP32-S2 and FT2232; JTAG debugging will then be possible using USB-UART/JTAG Port. See also [JTAG Debugging](#).
Breakout Header 2 | Some GPIO pins of the ESP32-S2-WROVER module are broken out to this header, see labels on the board.
USB-to-UART/JTAG Bridge | FT2232 adapter board allowing for communication over USB port using UART/JTAG protocols.
Camera Header | Mount a camera extension board here (e.g., ESP-LyraP-CAM).
Extension Header | Mount the extension boards having such connectors here.
Reset Button | Press this button to restart the system.
Boot Button | Holding down **Boot** and then pressing **Reset** initiates Firmware Download mode for downloading firmware through the serial port.
USB-UART/JTAG Port | Communication interface (UART or JTAG) between a PC and the ESP32-S2 module.
USB Power Port | Power supply for the board.
Battery Port | Connect an external battery to the 2-pin battery connector.
Power On LED | Turns on when the USB or an external power supply is connected to the board.
Power Switch | Switch to ON to power the system.
RGB Jumper | To have access to the RGB LED, place a jumper onto the pins.
RGB LED | Programmable RGB LED and controlled by GPIO45. Before using it, you need to put RGB Jumper ON.
Power Regulator | Regulator converts 5 V to 3.3 V.
I2C FPC Connector | (Reserved) Connect to other I2C extension boards using the FPC cable.
Breakout Header 1 | Some GPIO pins of the ESP32-S2-WROVER module are broken out to this header, see labels on the board.
Touch FPC Connector | Connect the ESP-LyraP-TouchA extension board using the FPC cable.
Touch Switch | In OFF position, GPIO1 to GPIO14 are used for connection to touch sensors; switch to ON if you want to use them for other purposes.
3.2” LCD FPC connector | Connect a 3.2” LCD extension board (e.g., ESP-LyraP-LCD32) using the FPC cable.

### Start Application Development
Before powering up your ESP32-S2-Kaluga-1, please make sure that it is in good condition with no obvious signs of damage.

### Required Hardware
- ESP32-S2-Kaluga-1
- Two USB 2.0 cables (Standard-A to Micro-B)
  - For power supply
  - For UART/JTAG communication
- Computer running Windows, Linux, or macOS
- Any extension boards of your choice

### Hardware Setup
1. Connect the extension board(s) of your choice (go to their respective user guides if necessary)
2. Plug in both USB cables
3. Turn the **Power Switch** to ON - the Power On LED will light up
**Software Setup** Please proceed to *Get Started*, where Section *Installation* will quickly help you set up the development environment.

The programming guide and application examples for your ESP32-S2-Kaluga-1 kit can be found in *esp-dev-kits* repository on GitHub.

**Contents and Packaging**

**Retail Orders** If you order one or several samples of the kit, each ESP32-S2-Kaluga-1 development kit comes in an individual package containing:

- **Main Board**
  - ESP32-S2-Kaluga-1
- **Extension Boards:**
  - ESP-LyraT-8311A
  - ESP-LyraP-CAM
  - ESP-LyraP-TouchA
  - ESP-LyraP-LCD32
- **Connectors**
  - 20-pin FPC cable (to connect ESP32-S2-Kaluga-1 to ESP-LyraP-TouchA)
- **Fasteners**
  - Mounting bolts (x8)
  - Screws (x4)
  - Nuts (x4)


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


**Hardware Reference**

**Block Diagram** A block diagram below shows the components of the ESP32-S2-Kaluga-1 and their interconnections.

**Power Supply Options** There are four ways to provide power to the board:

- Micro USB port, default power supply
- External battery via the 2-pin battery connector
- 5V and GND header pins
- 3V3 and GND header pins

**Compatibility of Extension Boards** If you want to use more than one extension board at the same time, please check the table given below.
### Boards Used

<table>
<thead>
<tr>
<th>Boards Used</th>
<th>HW Conflict</th>
<th>Limitations</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>8311A v1.2 + CAM v1.0</td>
<td>I2S Controller, IO46</td>
<td>ESP32-S2 has only one I2S interface. But both extension boards require connection via the ESP32-S2’s I2S interface (ESP-LyraT-8311A in Standard mode, ESP-LyraP-CAM in Camera mode). If IO46 is used by both extension boards at the same time, ESP-LyraP-CAM experiences interferences when used.</td>
<td>No ready solution for now.</td>
</tr>
<tr>
<td>TouchA v1.1 + LCD32 v1.1</td>
<td>IO11, IO6</td>
<td>Touch actions cannot be triggered because of the multiplexed pin IO11. ESP-LyraP-LCD32 is also affected because its BK (BLCT) pin is connected to pin IO6.</td>
<td>Do not initialize IO11 (NETWORK) and IO6 (PHOTO) for your ESP-LyraP-TouchA.</td>
</tr>
<tr>
<td>8311A v1.2 + LCD32 v1.1</td>
<td>IO6</td>
<td>The two extension boards can be used at the same time. However, since the BK (BLCT) pin of ESP32-S2-Kaluga-1 is connected to IO6, ESP-Lyra-T-8311A’s pin BT_ADC cannot be used and the board’s six buttons will not be available.</td>
<td>There is a solution that will allow you to use ESP-Lyra-T-8311A’s pin BT_ADC, but will stop you from controlling the display background brightness with software: on your ESP-LyraP-LCD32 board, remove R39, change R41 to 100 Ohm, switch BLCT_L to on.</td>
</tr>
<tr>
<td>TouchA v1.1 + 8311A v1.2</td>
<td>Pin BT_ADC on ESP-LyraT-8311A</td>
<td>The two extension boards can be used at the same time. However, ESP-LyraP-TouchA cannot be triggered if ESP-Lyra-T-8311A’s pin BT_ADC is used</td>
<td>If you plan to use ESP-Lyra-T-8311A’s pin BT_ADC, do not initialize pin IO6 (PHOTO) for your ESP-LyraP-TouchA.</td>
</tr>
</tbody>
</table>

---

*Fig. 22: ESP32-S2-Kaluga-1 block diagram*
Chapter 1. Get Started

Also, all extension boards and the JTAG interface share the same pins IO39, IO40, IO41 and IO42. For this reason, the following may disturb the JTAG operation:

- Plugging in any extension board
- Debugging an application that is using an extension board

### Known Issues

<table>
<thead>
<tr>
<th>Hardware Issue</th>
<th>Description</th>
<th>Reason for Failure</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP-LyraP-CAM v1.0, pin IO45, IO46</td>
<td>Flashing firmware might be impossible with the extension board connected to the main board.</td>
<td>Incorrect timing sequence is fed to strapping pins IO45 and IO46 when the board is powered on. It stops the board from booting successfully.</td>
<td>While flashing the main board, keep the extension board disconnected.</td>
</tr>
<tr>
<td>ESP-LyraP-CAM v1.0, pin IO45, IO46</td>
<td>Rebooting the board by pressing Reset might not lead to desired results.</td>
<td>Incorrect timing sequence is fed to strapping pins IO45 and IO46 when the board is powered on. It stops the board from booting successfully.</td>
<td>No ready solution for v1.2. This bug is fixed in ESP32-S2-Kaluga-1 V1.3.</td>
</tr>
<tr>
<td>ESP-LyraT-8311A v1.2, pin IO46</td>
<td>Flashing firmware might be impossible with the extension board connected to the main board.</td>
<td>Incorrect timing sequence is fed to strapping pin IO46 when the board is powered on. It stops the board from booting successfully.</td>
<td>While flashing the main board, keep the extension board disconnected.</td>
</tr>
<tr>
<td>ESP-LyraT-8311A v1.2, pin IO46</td>
<td>Rebooting the board by pressing Reset might not lead to desired results.</td>
<td>Incorrect timing sequence is fed to strapping pin IO46 when the board is powered on. It stops the board from booting successfully.</td>
<td>No ready solution for v1.2. This bug is fixed in ESP32-S2-Kaluga-1 V1.3.</td>
</tr>
</tbody>
</table>

### Hardware Revision Details

No previous versions available.

### Related Documents

**ESP-LyraP-CAM v1.0**

This user guide provides information on the ESP-LyraP-CAM extension board.

This extension board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as main boards below.

Currently, ESP-LyraP-CAM v1.0 is sold as part of the *ESP32-S2-Kaluga-1 Kit v1.2*.

The ESP-LyraP-CAM extends the functionality of your main board by adding a camera.

The document consists of the following major sections:

- **Overview**: Provides an overview and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP-LyraP-CAM’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraP-CAM.
- **Related Documents**: Gives links to related documentation.

**Overview**

This extension board adds a camera to your main board.
Chapter 1. Get Started

Fig. 23: ESP-LyraP-CAM

Fig. 24: ESP-LyraP-CAM - front and back
Description of Components

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Board Camera Header</td>
<td>Mount onto main board’s Camera Header</td>
</tr>
<tr>
<td>Power ON LED</td>
<td>Red LED is on if the power supply voltage is applied</td>
</tr>
<tr>
<td>Camera Module Connector</td>
<td>Supports OV2640 and OV3660 camera modules; this extension board is supplied</td>
</tr>
<tr>
<td></td>
<td>with an OV2640 camera module</td>
</tr>
<tr>
<td>Power Regulators</td>
<td>LDO Regulators converting 3.3 V to 2.8 V and 1.5 V</td>
</tr>
</tbody>
</table>

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

Required Hardware

- Board with a female Camera Header (e.g., ESP32-S2-Kaluga-1)
- ESP-LyraP-CAM extension board
- Computer running Windows, Linux, or macOS

Hardware Setup  Insert the ESP-LyraP-CAM extension board into your board’s female Camera Header.

Software Setup  See Section Software Setup of the ESP32-S2-Kaluga-1 kit user guide.

Hardware Reference

Block Diagram  A block diagram below shows the components of the ESP-LyraP-CAM and their interconnections.

![ESP-LyraP-CAM block diagram](image)

Fig. 25: ESP-LyraP-CAM block diagram

Hardware Revision Details  No previous versions available.

Related Documents

- ESP-LyraP-CAM Schematic (PDF)
- ESP-LyraP-CAM PCB Layout (PDF)

For other design documentation for the board, please contact us at sales@espressif.com.
Chapter 1. Get Started

**ESP-LyraP-LCD32 v1.1**

This user guide provides information on the ESP-LyraP-LCD32 extension board.

This extension board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as **main boards** below.

Currently, ESP-LyraP-LCD32 v1.1 is sold as part of the **ESP32-S2-Kaluga-1 Kit v1.2**.

The ESP-LyraP-LCD32 extends the functionality of your main board by adding an LCD graphic display.

The document consists of the following major sections:

- **Overview**: Provides an overview and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP-LyraP-LCD32’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraP-LCD32.
- **Related Documents**: Gives links to related documentation.

**Overview**  This extension board adds a 3.2” LCD graphic display with the resolution of 320x240. This display is connected to ESP32-S2 over the SPI bus.

**Description of Components**  In the description of components below, **Reserved** means that the functionality is available, but the current version of the kit does not use it.
**Key Component**  |  **Description**  
---|---
Extension Header  |  Male Extension Header for mounting onto a female Extension Header  
LCD display  |  This version has a 3.2” 320x240 SPI LCD display module; the display driver/controller is Sitronix ST7789V  
Touch Screen Switch  |  No support for touch screens, keep the switches to OFF to make the pins available for other uses  
Main Board 3.2” LCD FPC Connector  |  (Reserved) Connect to main board’s 3.2” LCD FPC connector  
Control Switch  |  Switch to ON to set Reset/Backlight_control/CS to default high or low; switch to OFF to make the pins available for other uses  

**Start Application Development**  Before powering up your ESP-LyraP-LCD32, please make sure that it is in good condition with no obvious signs of damage.

**Required Hardware**

- Board with a female Extension Header (e.g., ESP32-S2-Kaluga-1, ESP-LyraT-8311A)
- ESP-LyraP-LCD32 extension board
- Four mounting bolts (for stable mounting)
- Computer running Windows, Linux, or macOS

**Hardware Setup**  To mount your ESP-LyraP-LCD32 onto the board with a female Extension Header:

1. Install the four mounting bolts onto the board with a female Extension Header  
2. Align the ESP-LyraP-LCD32 with the bolts and Extension Header and insert it carefully

**Software Setup**  See Section *Software Setup* of the ESP32-S2-Kaluga-1 kit user guide.
Chapter 1. Get Started

Hardware Reference

**Block Diagram** A block diagram below shows the components of the ESP-LyraP-LCD32 and their interconnections.

![ESP-LyraP-LCD32 block diagram](image)

Fig. 28: ESP-LyraP-LCD32 block diagram

**Hardware Revision Details** No previous versions available.

Related Documents

- ESP-LyraP-LCD32 Schematic (PDF)
- ESP-LyraP-LCD32 PCB Layout (PDF)

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

ESP-LyraP-TouchA v1.1

This user guide provides information on the ESP-LyraP-TouchA extension board.

This board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as main boards below.

Currently, ESP-LyraP-TouchA v1.1 is sold as part of the following kits:

- ESP32-S2-Kaluga-1 Kit v1.3
- ESP32-S2-Kaluga-1 Kit v1.2

The ESP-LyraP-TouchA extends the functionality of your main board by adding touch buttons.

The document consists of the following major sections:

- **Overview**: Provides an overview and hardware setup instructions.
- **Hardware Reference**: Provides more detailed information about the ESP-LyraP-TouchA’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraP-TouchA.
- **Related Documents**: Gives links to related documentation.

**Overview** The ESP-LyraP-TouchA has six touch buttons and is mainly designed for audio applications. However, the touch buttons can also be used for any other purposes.
Fig. 29: ESP-LyraP-TouchA

Fig. 30: ESP-LyraP-TouchA
Description of Components

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Board Touch FPC Connector</td>
<td>Connect to main board’s Touch FPC Connector.</td>
</tr>
<tr>
<td>Touchpad</td>
<td>Capacitive touch electrode.</td>
</tr>
<tr>
<td>Guard Ring</td>
<td>Connected to a touch sensor, the guard ring triggers an interrupt if wet (Water rejection). It indicates that the sensor array is also wet and most (if not all) touchpads are unusable due to the false detection of touches. After receiving this interrupt, the user might consider disabling all the touch sensors via software.</td>
</tr>
</tbody>
</table>

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

Required Hardware

- Board with a Touch FPC connector (e.g., ESP32-S2-Kaluga-1)
- ESP-LyraP-TouchA extension board
- FPC cable
- Computer running Windows, Linux, or macOS

Hardware Setup  Connect the two FPC connectors with the FPC cable.

Software Setup  See Section Software Setup of the ESP32-S2-Kaluga-1 kit user guide.

Hardware Reference

Block Diagram  A block diagram below shows the components of ESP-LyraP-TouchA and their interconnections.

![Block Diagram](image-url)

Fig. 31: ESP-LyraP-TouchA-v1.1 block diagram

Hardware Revision Details  No previous versions available.
Related Documents

- ESP-LyraP-TouchA Schematic (PDF)
- ESP-LyraP-TouchA PCB Layout (PDF)

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

ESP-LyraT-8311A v1.2

This user guide provides information on the ESP-LyraT-8311A extension board.

This board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as main boards below.

Currently, ESP-LyraT-8311A v1.2 is sold as part of the ESP32-S2-Kaluga-1 Kit v1.2.

The ESP-LyraT-8311A extends the functionality of your main board by adding sound processing functionality:

- Audio playback/recording
- Processing of audio signals
- Programmable buttons for easy control

This extension board can be used in many ways. The applications might include voice user interface, voice control, voice authorization, recording and playback of sound, etc.

Fig. 32: ESP-LyraT-8311A (click to enlarge)

The document consists of the following major sections:

- **Overview**: Provides an overview and hardware setup instructions.
- **Hardware reference**: Provides more detailed information about the ESP-LyraT-8311A’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraT-8311A.
- **Related Documents**: Gives links to related documentation.
Overview  The ESP-LyraT-8311A is mainly designed for audio applications. However, you can use your creativity to come up with any other use cases.

![ESP-LyraT-8311A-front](image)

**Fig. 33: ESP-LyraT-8311A - front (click to enlarge)**

**Description of Components**  The description of components starts from the top right corner and then goes clockwise.

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension Header</td>
<td>Male Extension Header on the flip side is for mounting onto main board’s Extension Header; Female Extension Header is for mounting other boards that have a Male Extension Header</td>
</tr>
<tr>
<td>Function Button</td>
<td>This board has six programmable buttons</td>
</tr>
<tr>
<td>Microphone</td>
<td>Supports Electret and MEMS microphones; this extension board is supplied with an electret microphone</td>
</tr>
<tr>
<td>Headphone Connector</td>
<td>6.3 mm (1/8”) stereo headphone connector</td>
</tr>
<tr>
<td>Speaker Connector</td>
<td>Connect an external speaker to the 2-pin connector</td>
</tr>
<tr>
<td>PA</td>
<td>3 W Audio signal amplifier for the external speaker</td>
</tr>
<tr>
<td>External Mic-Matrix Connector</td>
<td>(Reserved) FPC connector for external Mic-Matrix (microphone boards)</td>
</tr>
<tr>
<td>ADC</td>
<td>(Reserved) high-performance ADC/ES7243: 1 channel for microphone, 1 channel for acoustic echo cancellation (AEC) function</td>
</tr>
<tr>
<td>Mono Audio Codec</td>
<td>ES8311 audio ADC and DAC; it can convert the analog signal picked up by the microphone or convert digital signal to play it back through a speaker or headphones</td>
</tr>
</tbody>
</table>

Start Application Development  Before powering up your ESP-LyraT-8311A, please make sure that it is in good condition with no obvious signs of damage.

**Required Hardware**
Chapter 1. Get Started

- Board with a female Extension Header (e.g., ESP32-S2-Kaluga-1)
- ESP-LyraT-8311A extension board
- Four mounting bolts (for stable mounting)
- Computer running Windows, Linux, or macOS

**Hardware Setup**  To mount your ESP-LyraT-8311A onto the board with a female Extension Header:

1. Install the four mounting bolts onto the board with a female Extension Header
2. Align the ESP-LyraT-8311A with the bolts and Extension Header and insert it carefully

**Software Setup**  Depending on your application, see:

- ESP-ADF Getting Started Guide if you develop with ESP-ADF (Espressif Audio Development Framework).
- Section Software Setup of the ESP32-S2-Kaluga-1 kit user guide if you develop directly with ESP-IDF (Espressif IOT Development Framework).

**Hardware Reference**

**Block Diagram**  A block diagram below shows the components of ESP-LyraT-8311A and their interconnections.

![ESP-LyraT-8311A block diagram](image)

Fig. 34: ESP-LyraT-8311A block diagram

**Hardware Revision Details**  No previous versions available.

**Related Documents**

- ESP-LyraT-8311A Schematic (PDF)
- ESP-LyraT-8311A PCB Layout (PDF)

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

- ESP32-S2-WROVER Datasheet (PDF)
- ESP Product Selector
- JTAG Debugging
- ESP32-S2-Kaluga-1 Schematic (PDF)
• ESP32-S2-Kaluga-1 PCB Layout (PDF)
• ESP32-S2-Kaluga-1 Pin Mapping (Excel)

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

**ESP-LyraP-CAM v1.1**

This user guide provides information on the ESP-LyraP-CAM extension board.

This extension board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as main boards below.

Currently, ESP-LyraP-CAM v1.1 is sold as part of the **ESP32-S2-Kaluga-1 Kit v1.3**.

The ESP-LyraP-CAM extends the functionality of your main board by adding a camera.

![ESP-LyraP-CAM](image)

**Fig. 35: ESP-LyraP-CAM**

The document consists of the following major sections:

- **Overview**: Provides an overview and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP-LyraP-CAM’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraP-CAM.
- **Related Documents**: Gives links to related documentation.

**Overview**  This extension board adds a camera to your main board.
Chapter 1. Get Started

Fig. 36: ESP-LyraP-CAM - front and back

### Description of Components

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Board Camera Header</td>
<td>Mount onto main board’s Camera Header</td>
</tr>
<tr>
<td>Power ON LED</td>
<td>Red LED is on if the power supply voltage is correct</td>
</tr>
<tr>
<td>Camera Module Connector</td>
<td>Supports OV2640 and OV3660 camera modules; this extension board is supplied with an OV2640 camera module</td>
</tr>
<tr>
<td>Power Regulators</td>
<td>LDO Regulators converting 3.3 V to 2.8 V and 1.5 V</td>
</tr>
</tbody>
</table>

### Start Application Development
Before powering up your ESP-LyraP-CAM, please make sure that it is in good condition with no obvious signs of damage.

### Required Hardware
- Board with a female Camera Header (e.g., ESP32-S2-Kaluga-1)
- ESP-LyraP-CAM extension board
- Computer running Windows, Linux, or macOS

### Hardware Setup
Insert the ESP-LyraP-CAM extension board into your board’s female Camera Header.

### Software Setup
See Section Software Setup of the ESP32-S2-Kaluga-1 kit user guide.

### Hardware Reference

**Block Diagram** A block diagram below shows the components of the ESP-LyraP-CAM and their interconnections.

**Hardware Revision Details**
Chapter 1. Get Started

ESP-LyraP-CAM v1.1

- Silk screen updated
- No actual hardware updates

ESP-LyraP-CAM v1.0  *Initial release*

Related Documents

- ESP-LyraP-CAM Schematic (PDF)
- ESP-LyraP-CAM PCB Layout (PDF)

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

ESP-LyraP-LCD32 v1.2

This user guide provides information on the ESP-LyraP-LCD32 extension board.

This extension board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as *main boards* below.

Currently, ESP-LyraP-LCD32 v1.2 is sold as part of the *ESP32-S2-Kaluga-1 Kit v1.3*.

The ESP-LyraP-LCD32 extends the functionality of your main board by adding an LCD graphic display.

The document consists of the following major sections:

- **Overview**: Provides an overview and hardware/software setup instructions to get started.
- **Hardware reference**: Provides more detailed information about the ESP-LyraP-LCD32’s hardware.
- **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraP-LCD32.
- **Related Documents**: Gives links to related documentation.

**Overview**  This extension board adds a 3.2” LCD graphic display with the resolution of 320x240. This display is connected to ESP32-S2 over the SPI bus.

**Description of Components**  In the description of components below, *Reserved* means that the functionality is available, but the current version of the kit does not use it.
Chapter 1. Get Started

Fig. 38: ESP-LyraP-LCD32 (click to enlarge)

Fig. 39: ESP-LyraP-LCD32 - front (click to enlarge)
Chapter 1. Get Started

Key Component | Description
--- | ---
Extension Header | Male Extension Header for mounting onto a female Extension Header
LCD Display | This version has a 3.2” 320x240 SPI LCD display module; the display driver/controller is either Sitronix ST7789V or Ilitek ILI9341
Touch Screen Switch | No support for touch screens, keep the switches to OFF to make the pins available for other uses
Main Board 3.2” LCD FPC Connector | (Reserved) Connect to main board’s 3.2” LCD FPC connector
Control Switch | Switch to ON to set Reset/Backlight_control/CS to default high or low; switch to OFF to make the pins available for other uses

Start Application Development  Before powering up your ESP-LyraP-LCD32, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- Board with a female Extension Header (e.g., ESP32-S2-Kaluga-1, ESP-LyraT-8311A)
- ESP-LyraP-LCD32 extension board
- Four mounting bolts (for stable mounting)
- Computer running Windows, Linux, or macOS

Hardware Setup  To mount your ESP-LyraP-LCD32 onto the board with a female Extension Header:

1. Install the four mounting bolts onto the board with a female Extension Header
2. Align the ESP-LyraP-LCD32 with the bolts and Extension Header and insert it carefully

Software Setup  See Section Software Setup of the ESP32-S2-Kaluga-1 kit user guide.
Hardware Reference

**Block Diagram**  A block diagram below shows the components of the ESP-LyraP-LCD32 and their interconnections.

![ESP-LyraP-LCD32 block diagram](image)

**Hardware Revision Details**

**ESP-LyraP-LCD32 v1.2**
- LCD backlight default ON, cannot be controlled by MCU
- Touch Driver and related switch removed for major limitations caused by multiplexed pins

**ESP-LyraP-LCD32 v1.1  Initial release**

**Related Documents**
- ESP-LyraP-LCD32 Schematic (PDF)
- ESP-LyraP-LCD32 PCB Layout (PDF)

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

**ESP-LyraT-8311A v1.3**

This user guide provides information on the ESP-LyraT-8311A extension board.

This board cannot be bought separately and is usually sold together with other Espressif development boards (e.g., ESP32-S2-Kaluga-1), which will be referred to as main boards below.

Currently, ESP-LyraT-8311A v1.3 is sold as part of the ESP32-S2-Kaluga-1 Kit v1.3.

The ESP-LyraT-8311A extends the functionality of your main board by adding sound processing functionality:
- Audio playback/recording
- Processing of audio signals
- Programmable buttons for easy control

This extension board can be used in many ways. The applications might include voice user interface, voice control, voice authorization, recording and playback of sound, etc.

The document consists of the following major sections:
- **Overview**: Provides an overview and hardware setup instructions.
Fig. 42: ESP-LyraT-8311A (click to enlarge)
• **Hardware reference**: Provides more detailed information about the ESP-LyraT-8311A’s hardware.
• **Hardware Revision Details**: Covers revision history, known issues, and links to user guides for previous versions of the ESP-LyraT-8311A.
• **Related Documents**: Gives links to related documentation.

**Overview**  The ESP-LyraT-8311A is mainly designed for audio applications. However, you can use your creativity to come up with any other use cases.

![ESP-LyraT-8311A front](image)

**Description of Components**  The description of components starts from the top right corner and then goes clock-wise.

**Reserved** means that the functionality is available, but the current version of the kit does not use it.

<table>
<thead>
<tr>
<th>Key Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension Header</td>
<td>Male Extension Header on the flip side is for mounting onto main board’s Extension Header; Female Extension Header is for mounting other boards that have a Male Extension Header</td>
</tr>
<tr>
<td>Function Button</td>
<td>This board has six programmable buttons</td>
</tr>
<tr>
<td>Microphone</td>
<td>Supports Electret and MEMS microphones; this extension board is supplied with an electret microphone</td>
</tr>
<tr>
<td>Headphone Connector</td>
<td>6.3 mm (1/8”) stereo headphone connector</td>
</tr>
<tr>
<td>Speaker Connector</td>
<td>Connect an external speaker to the 2-pin connector</td>
</tr>
<tr>
<td>PA</td>
<td>3 W Audio signal amplifier for the external speaker</td>
</tr>
<tr>
<td>External Mic-Matrix Connector</td>
<td>(Reserved) FPC connector for external Mic-Matrix (microphone boards)</td>
</tr>
<tr>
<td>ADC</td>
<td>(Reserved) high-performance ADC/ES7243: 1 channel for microphone, 1 channel for acoustic echo cancellation (AEC) function</td>
</tr>
<tr>
<td>Mono Audio Codec</td>
<td>ES8311 audio ADC and DAC; it can convert the analog signal picked up by the microphone or convert digital signal to play it back through a speaker or headphones</td>
</tr>
</tbody>
</table>
Chapter 1. Get Started

**Start Application Development** Before powering up your ESP-LyraT-8311A, please make sure that it is in good condition with no obvious signs of damage.

**Required Hardware**
- Board with a female Extension Header (e.g., ESP32-S2-Kaluga-1)
- ESP-LyraT-8311A extension board
- Four mounting bolts (for stable mounting)
- Computer running Windows, Linux, or macOS

**Hardware Setup** To mount your ESP-LyraT-8311A onto the board with a female Extension Header:
1. Install the four mounting bolts onto the board with a female Extension Header
2. Align the ESP-LyraT-8311A with the bolts and Extension Header and insert it carefully

**Software Setup** Depending on your application, see:
- ESP-ADF Getting Started Guide if you develop with ESP-ADF (Espressif Audio Development Framework).
- Section Software Setup of the ESP32-S2-Kaluga-1 kit user guide if you develop directly with ESP-IDF (Espressif IOT Development Framework).

**Hardware Reference**

**Block Diagram** A block diagram below shows the components of ESP-LyraT-8311A and their interconnections.

![ESP-LyraT-8311A block diagram](image)

Fig. 44: ESP-LyraT-8311A block diagram

**Hardware Revision Details**

**ESP-LyraT-8311A v1.3**
- ADC/ES7243 and ADC/ES7210 removed as the Mono Audio Codec chip provides all the needed functionality.

**ESP-LyraT-8311A v1.2** *Initial release*
Chapter 1. Get Started

Related Documents

- ESP-LyraT-8311A Schematic (PDF)
- ESP-LyraT-8311A PCB Layout (PDF)

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

- ESP32-S2-WROVER Datasheet (PDF)
- ESP Product Selector
- JTAG Debugging
- ESP32-S2-Kaluga-1 Schematic (PDF)
- ESP32-S2-Kaluga-1 PCB Layout (PDF)
- ESP32-S2-Kaluga-1 Pin Mapping (Excel)

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

1.2.2 Software

To start using ESP-IDF on ESP32-S2, install the following software:

- **Toolchain** to compile code for ESP32-S2
- **Build tools** - CMake and Ninja to build a full **Application** for ESP32-S2
- **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32-S2 and scripts to operate the **Toolchain**

![Software Diagram]

1.3 Installation

To install all the required software, we offer some different ways to facilitate this task. Choose from one of the available options.
Chapter 1. Get Started

1.3.1 IDE

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

- Eclipse Plugin
- VSCode Extension

1.3.2 Manual Installation

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

**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 are going to use the Command Prompt, but after ESP-IDF is installed you can use Eclipse Plugin or another graphical IDE with CMake support instead.

**Note:** Limitations: - The installation path of ESP-IDF and ESP-IDF Tools must not be longer than 90 characters. Too long installation paths might result in a failed build. - The installation path of Python or ESP-IDF must not contain white spaces or parentheses. - 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 > 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.

What Is the Use case for Online and Offline Installer Online Installer is very small and allows the installation of all available releases of ESP-IDF. The installer downloads 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 launches ESP-IDF environment in selected prompt.

Run ESP-IDF PowerShell Environment:

![Fig. 45: Completing the ESP-IDF Tools Setup Wizard with Run ESP-IDF PowerShell Environment](image)

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

Using the Command Prompt  For the remaining Getting Started steps, we are 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.

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
Chapter 1. Get Started

Fig. 46: ESP-IDF PowerShell

Fig. 47: Completing the ESP-IDF Tools Setup Wizard with Run ESP-IDF Command Prompt (cmd.exe)
Chapter 1. Get Started

Fig. 48: ESP-IDF Command Prompt

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 2.38.0.windows.1
Setting IDF_PATH: C:\Users\test\esp\esp-idf

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

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

This guide helps you on the first steps using ESP-IDF. Follow this guide to start a new project on the ESP32-S2 and build, flash, and monitor the device output.

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

Start a Project  
Now you are ready to prepare your application for ESP32-S2. You can start with `get-started/hello_world` project from `examples` directory in ESP-IDF.

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

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

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

Note: 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.

Connect Your Device  
Now connect your ESP32-S2 board to the computer and check under which serial port the board is visible.

Serial port names start with `COM` in Windows.

If you are not sure how to check the serial port name, please refer to Establish Serial Connection with ESP32-S2 for full details.

Note: Keep the port name handy as it is needed in the next steps.

Configure Your Project  
Navigate to your `hello_world` directory, set ESP32-S2 as the target, and run the project configuration utility `menuconfig`.

Windows

```
cd %userprofile%\esp\hello_world
idf.py set-target esp32s2
idf.py menuconfig
```

After opening a new project, you should first set the target with `idf.py set-target esp32s2`. Note that existing builds and configurations in the project, if any, are cleared and initialized in this process. The target may
be saved in the environment variable to skip this step at all. See *Select the Target Chip: set-target* for additional information.

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

![Project configuration - Home window](image)

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

**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.

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

To use the USB for flashing the ESP32-S2, you need to change the channel for the console output to USB. For the ESP32-S2, the default console output channel is the UART:

1. Navigate to the option **Channel for console output**.
   
   Component config ---> ESP System Settings ---> Channel for console output
2. Change to the option (the default is always UART):
   
   USB CDC
3. Save the new configuration and exit the menuconfig screen.

**Build the Project**  Build the project by running:

```
idf.py build
```

This command compiles the application and all ESP-IDF components, then it generates 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")
```

(continues on next page)
Chapter 1. Get Started

(continued from previous page)

-- Building empty aws_iot component due to configuration
-- Component names: ...
-- Component paths: ...
...
[527/527] Generating hello_world.bin
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 finishes by generating the firmware binary .bin files.

Flash onto the Device  To flash the binaries that you just built for the ESP32-S2 in the previous step, you need to run the following command:

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

Replace PORT with your ESP32-S2 board’s USB port name. If the PORT is not defined, the `idf.py` will try to connect automatically using the available USB ports.

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.

Encountered Issues While Flashing? See the “Additional Tips” below. You can also refer to Flashing Troubleshooting page or Establish Serial Connection with ESP32-S2 for more detailed information.

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

```
...
esptool.py --chip esp32s2 -p /dev/ttyUSB0 -b 460800 --before=default_reset --
--after=hard_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB--
--0x8000 partition_table/partition-table.bin 0x1000 bootloader/bootloader.bin--
--0x10000 hello_world.bin
esptool.py v3.0-dev
Serial port /dev/ttyUSB0
Connecting....
Chip is ESP32-S2
Features: WiFi
Crystal is 40MHz
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.0 seconds (effective 3851.6...
--kbit/s)... Hash of data verified.
```
If there are no issues by the end of the flash process, the board will reboot and start up the "hello_world" application.

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

**Monitor the Output** 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 <PORT> 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 <PORT> 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)
est 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...  
This is esp32s2 chip with 1 CPU core(s), WiFi, silicon revision 0, 2 MB...
Minimum free heap size: 253900 bytes  
Restarting in 9 seconds...  
Restarting in 8 seconds...  
Restarting in 7 seconds...```

To exit IDF monitor use the shortcut Ctrl+].

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

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

See also:
Chapter 1. Get Started

- **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 is all that you need to get started with ESP32-S2!**

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

**Important:** Some of examples do not support ESP32-S2 because required hardware is not included in ESP32-S2 so it cannot be supported.

If building an example, please check the README file for the **Supported Targets** table. If this is present including ESP32-S2 target, or the table does not exist at all, the example will work on ESP32-S2.

### Additional Tips

**Permission Denied Issue** With some Linux distributions, you may get the error message similar to `Could not open port <PORT>: Permission denied: '<PORT>'` when flashing the ESP32-S2. *This can be solved by adding the current user to the specific group*, such as dialout or uucp group.

**Python Compatibility** ESP-IDF supports Python 3.8 or newer. It is recommended to upgrade your operating system to a recent version satisfying this requirement. Other options include the installation of Python from sources or the use of a Python version management system such as pyenv.

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

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

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

Here is an example of how to add ESP32-S2-Kaluga-Kit BSP to your project:

```
idf.py add-dependency esp32_s2_kaluga_kit
```

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

**Flash Erase** Erasing the flash is also possible. To erase the entire flash memory you can run the following command:

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

For erasing the OTA data, if present, you can run this command:

```
idf.py -p PORT erase-otadata
```

The flash erase command can take a while to be done. Do not disconnect your device while the flash erasing is in progress.

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

- *Updating ESP-IDF Tools on Windows*
- *Establish Serial Connection with ESP32-S2*
- Eclipse Plugin
- VSCode Extension
- **IDF Monitor**
Chapter 1. Get Started

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 downloads and installs 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 was not 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`:

```
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`:

```
export.ps1
```

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

Establish Serial Connection with ESP32-S2

Establishing a serial connection with the ESP32-S2 target device could be done using USB-to-UART bridge or USB peripheral supported in ESP32-S2.

Some development boards have the USB-to-UART bridge installed. If a board does not have a bridge then an external bridge may be used.

**Supported USB Peripheral**  The ESP32-S2 supports the USB peripheral. In this case, the USB-to-UART bridge is not needed and the device can be flashed directly.

Apart from the USB peripheral, some development boards also include the USB-to-UART bridge.

**USB-to-UART Bridge on Development Board**  For boards with an installed USB-to-UART bridge, the connection between the personal computer and the bridge is USB and between the bridge and ESP32-S2 is UART.

**External USB-to-UART Bridge**  Sometimes the USB-to-UART bridge is external. This is often used in small development boards or finished products when space and costs are crucial.
Fig. 50: SoC with Supported USB

Fig. 51: Development Board with USB-to-UART Bridge

Fig. 52: External USB-to-UART Bridge
Chapter 1. Get Started

**Flash Using USB**  For the ESP32-S2, the USB peripheral is available, allowing you to flash the binaries without the need for an external USB-to-UART bridge.

The USB on the ESP32-S2 uses the **GPIO20** for **D+** and **GPIO19** for **D-**.

After flashing the binaries, a manual reset is needed.

**Flash Using UART**  This section provides guidance on how to establish a serial connection between ESP32-S2 and PC using USB-to-UART Bridge, either installed on the development board or external.

**Connect ESP32-S2 to PC**  Connect the ESP32-S2 board to the PC using the USB cable. If device driver does not install automatically, identify USB-to-UART bridge on your ESP32-S2 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 ESP32-S2 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-UART bridge 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.

For devices downloaded using a USB-to-UART bridge, you can run the following command including the optional argument to define the baud rate.

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

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

**Note:** If the device does not support the auto download mode, you need to get into the download mode manually. To do so, press and hold the **BOOT** button and then press the **RESET** button once. After that release the **BOOT** button.

**Check Port on Windows**  Check the list of identified COM ports in the Windows Device Manager. Disconnect ESP32-S2 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-S2 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**

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

**macOS**

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

**Note:** macOS users: if you do not see the serial port then check you have the USB/serial drivers installed. See Section **Connect ESP32-S2 to PC** for links to drivers. 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.
Fig. 53: USB to UART bridge of ESP32-DevKitC in Windows Device Manager
Fig. 54: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager
Adding User to dialout or uucp 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 by checking if you get any output on the terminal after resetting ESP32-S2.

The default console baud rate on ESP32-S2 is 115200.

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

Run terminal and set identified serial port. Baud rate = 115200 (if needed, change this to the default baud rate of the chip in use), data bits = 8, stop bits = 1, and parity = N. Below are example screenshots 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. 55: Setting Serial Communication in PuTTY on Windows](image)

Then open serial port in terminal and check, if you see any log printed out by ESP32-S2. The log contents depend on application loaded to ESP32-S2, see Example Output. Reset the board if no log has been printed out.
Fig. 56: Setting Serial Communication in PuTTY on Linux
Note: Close the serial terminal after verification that communication is working. If you keep the terminal session open, the serial port will be inaccessible for uploading firmware later.

Note: If there is no log output, check

- if the required power is supplied to ESP32-S2
- if the board was reset after starting the terminal program
- if the selected serial port is the correct one by using the method stated in Check Port on Windows and Check Port on Linux and macOS
- if the serial port is not being used by another program
- if the identified port has been selected in serial terminal programs you are using, as stated in Windows and Linux
- if settings of the serial port in serial terminal programs are applicable to corresponding applications
- if the correct USB connector (UART) is used on the development board
- if your application is expected to output some log
- if the log output has not been disabled (use hello world application to test)

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

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

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

- You should see similar output:

```
/dev/cu.Bluetooth-Incoming-Port /dev/cu.SLAB_USBtoUART /dev/cu.SLAB_USBtoUART7
```

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

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

Replace `device_name` with the name found running `ls /dev/cu.*`.
- What you are looking for is some log displayed by the `screen`. The log contents depend on application loaded to ESP32-S2, see Example Output. To exit the current screen session, type Ctrl-A + K.

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

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

```
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)
configsp: 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
```
If you can see readable log output, it means serial connection is working and you are ready to proceed with installation and finally upload an application to ESP32-S2.

**Note:** For some serial port wiring configurations, the serial RTS & DTR pins need to be disabled in the terminal program before the ESP32-S2 booting and producing 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](https://docs.espressif.com/projects/esp-idf/en/latest/esp-idf/reference Manual.html) for more details.

If you got here from *Step 5. First Steps on ESP-IDF* when installing s/w for ESP32-S2 development, then you can continue with *Step 5. First Steps on ESP-IDF*.

**Flashing Troubleshooting**

**Failed to Connect** If you run the given command and see errors such as “Failed to connect”, there might be several reasons for this. One of the reasons might be issues encountered by `esptool.py`, the utility that is called by the build system to reset the chip, interact with the ROM bootloader, and flash firmware. One simple solution to try is to manually reset as described below. If it does not help, you can find more details about possible issues in the `esptool troubleshooting` page.

`esptool.py` resets ESP32-S2 automatically by asserting DTR and RTS control lines of the USB-to-UART bridge, i.e., FTDI or CP210x (for more information, see *Establish Serial Connection with ESP32-S2*). The DTR and RTS control lines are in turn connected to GPIO0 and CHIP_PU (EN) pins of ESP32-S2, thus changes in the voltage levels of DTR and RTS will boot ESP32-S2 into Firmware Download mode. As an example, check the [schematic](https://docs.espressif.com/projects/esp-idf/en/latest/esp-idf/reference Manual.html) for the ESP32 DevKitC development board.

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

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

Depending on the kind of hardware you have, it may also be possible to manually put your ESP32-S2 board into Firmware Download mode (reset).

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

**IDF Monitor**

IDF Monitor uses the `esp-idf-monitor` package as a serial terminal program which relays serial data to and from the target device’s serial port. It also provides some ESP-IDF-specific features.

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

**Keyboard Shortcuts** For easy interaction with IDF Monitor, use the keyboard shortcuts given in the table. These keyboard shortcuts can be customized, for more details see *Configuration File* section.
### Chapter 1. Get Started

<table>
<thead>
<tr>
<th>Keyboard Shortcut</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + ]</td>
<td>Exit the program</td>
<td></td>
</tr>
<tr>
<td>Ctrl + T</td>
<td>Menu escape key</td>
<td>Press and follow it by one of the keys given below.</td>
</tr>
<tr>
<td>• Ctrl + T</td>
<td>Send the menu character itself to remote</td>
<td></td>
</tr>
<tr>
<td>• Ctrl + J</td>
<td>Send the exit character itself to remote</td>
<td></td>
</tr>
<tr>
<td>• Ctrl + P</td>
<td>Reset target into bootloader to pause app via RTS line</td>
<td>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.</td>
</tr>
<tr>
<td>• Ctrl + R</td>
<td>Reset target board via RTS</td>
<td>Resets the target board and re-starts the application via the RTS line (if connected).</td>
</tr>
<tr>
<td>• Ctrl + F</td>
<td>Build and flash the project</td>
<td>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).</td>
</tr>
<tr>
<td>• Ctrl + A (or A)</td>
<td>Build and flash the app only</td>
<td>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).</td>
</tr>
<tr>
<td>• Ctrl + Y</td>
<td>Stop/resume log output printing on screen</td>
<td>Discards all incoming serial data while activated. Allows to quickly pause and examine log output without quitting the monitor.</td>
</tr>
<tr>
<td>• Ctrl + L</td>
<td>Stop/resume log output saved to file</td>
<td>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).</td>
</tr>
<tr>
<td>• Ctrl + I (or I)</td>
<td>Stop/resume printing timestamps</td>
<td>IDF Monitor can print a timestamp in the beginning of each line. The timestamp format can be changed by the --timestamp-format command line argument.</td>
</tr>
<tr>
<td>• Ctrl + H (or H)</td>
<td>Display all keyboard shortcuts</td>
<td></td>
</tr>
<tr>
<td>• Ctrl + X (or X)</td>
<td>Exit the program</td>
<td></td>
</tr>
<tr>
<td>Ctrl + C</td>
<td>Interrupt running application</td>
<td>Pauses IDF Monitor and runs GDB project debugger to debug the application at runtime. This requires CONFIG_ESP_SYSTEM_GDBSTUB_RUNTIME option to be enabled.</td>
</tr>
</tbody>
</table>

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

### ESP-IDF-specific Features

#### Automatic Address Decoding
Whenever the chip outputs a hexadecimal address that points to executable code, IDF monitor looks up the location in the source code (file name and line number) and prints the location on the next line in yellow.

If an ESP-IDF app crashes and panics, a register dump and backtrace are produced, such as the following:
Chapter 1. Get Started

Guru Meditation Error of type StoreProhibited occurred on core 0. Exception was unhandled.

Register dump:
PC : 0x400f360d PS : 0x00060330 A0 : 0x800dbf56 A1 :...
-0x3ff7e00
A2 : 0x3ffb136c A3 : 0x00000005 A4 : 0x00000000 A5 :...
-0x00000000
A6 : 0x00000000 A7 : 0x00000080 A8 : 0x00000000 A9 :...
-0x3ff7dd0
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

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 :...
-0x3ff7e00
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 :...
-0x3ff7dd0
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:52
0x400dbf5e: dont_crash at /home/gus/esp/32/idf/examples/get-started/hello_world/main/./hello_world_main.c:47
0x400dbf82: app_main at /home/gus/esp/32/idf/examples/get-started/hello_world/main/./hello_world_main.c:42
0x400d071d: main_task at /home/gus/esp/32/idf/components/esp32s2/.cpu_start.c:254

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

```
xtensa-esp32s2-elf-addr2line -pfiaC -e build/PROJECT.elf ADDRESS
```

If an address is not matched in the app source code, IDF monitor also checks the ROM code. Instead of printing the source file name and line number, only the function name followed by in ROM is displayed:
abort() was called at PC 0x40007c69 on core 0
0x40007c69: ets_write_char in ROM

Backtrace:
0x40081656:0x3ffb4ac0 0x40085729:0x3ffb4ae0 0x4008a7ce:0x3ffb4b00...
→ 0x40007c69:0x3ffb4b70 0x40008148:0x3ffb4b90 0x400d51d7:0x3ffb4c20...
→ 0x400e31bc:0x3ffb4c50 0x40087bc5:0x3ffb4c80
0x40081656: panic_abort at /Users/espressif/esp-idf/components/esp_system/panic.c:452
0x40085729: esp_system_abort at /Users/espressif/esp-idf/components/esp_system/port/esp_system_chip.c:90
0x4008a7ce: abort at /Users/espressif/esp-idf/components/newlib/abort.c:38
0x40007c69: ets_write_char in ROM
0x40008148: ets_printf in ROM
0x400d51d7: app_main at /Users/espressif/esp-idf/examples/get-started/hello_world/
→ main/hello_world_main.c:49
0x400e31bc: main_task at /Users/espressif/esp-idf/components/freertos/app_startup.c:
→ c:208 (discriminator 13)
0x40087bc5: vPortTaskWrapper at /Users/espressif/esp-idf/components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c:162

The ROM ELF file is automatically loaded from a location based on the IDF_PATH and ESP_ROM_ELF_DIR environment variables. This can be overridden by calling esp_idf_monitor and providing a path to a specific ROM ELF file:

```
python -m esp_idf_monitor --rom-elf-file [path to ROM ELF file]
```

**Note:** Set environment variable ESP_MONITOR_DECODE to 0 or call esp_idf_monitor with specific command line option: python -m esp_idf_monitor --disable-address-decoding to disable address decoding.

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

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

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

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

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

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

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

```
```
xtensa-esp32s2-elf-gdb -ex "set serial baud BAUD" -ex "target remote PORT" -ex...

### 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. Filtering can also be configured using the `ESP_IDF_MONITOR_PRINT_FILTER` environment variable.

**Note:** When using both the environment variable `ESP_IDF_MONITOR_PRINT_FILTER` and the argument `--print-filter`, the setting from the CLI argument will take precedence.

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 a Verbose level.

**Note:** Use primary logging to disable at compilation the outputs you do not need through the `logging library`. Output filtering with the IDF monitor is a secondary solution that 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:

```plaintext
load:0x40078000,len:13564
entry 0x40078d4c
E (31) esp_image: image at 0x300000 has invalid magic byte
W (31) esp_image: image at 0x300000 has invalid SPI mode 255
```

(continues on next page)
Chapter 1. Get Started

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
D (318) vfs: esp vfs_register_fd_range is successful for range <54; 64) and VFS ID_ ~1
I (328) wifi: wifi driver task: 3ffdf84, 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: 3ffdf84, 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
eentry 0x40078d4c
I (569) heap_init: Initializing. RAM available for dynamic allocation:
D (309) light_driver: [light_init, 74]:status: 1, mode: 2

Configuration File  esp-idf-monitor is using C0 control codes to interact with the console. Characters from the config file are converted to their C0 control codes. Available characters include the English alphabet (A-Z) and special symbols: [,], , ^, _.

Warning: Please note that some characters may not work on all platforms or can be already reserved as a shortcut for something else. Use this feature with caution!

File Location  The default name for a configuration file is esp-idf-monitor.cfg. First, the same directory esp-idf-monitor is being run if is inspected.

If a configuration file is not found here, the current user’s OS configuration directory is inspected next:
- Linux: /home/<user>/.config/esp-idf-monitor/
- MacOS /Users/<user>/.config/esp-idf-monitor/
- Windows: C:\Users\<user>\AppData\Local\esp-idf-monitor\n
If a configuration file is still not found, the last inspected location is the home directory:
- Linux: /home/<user> /
- MacOS /Users/<user> /
- Windows: C:\Users\<user>\n
On Windows, the home directory can be set with the HOME or USERPROFILE environment variables. Therefore, the Windows configuration directory location also depends on these.

A different location for the configuration file can be specified with the ESP_IDF_MONITOR_CFGFILE environment variable, e.g., ESP_IDF_MONITOR_CFGFILE = ~/custom_config.cfg. This overrides the search priorities described above.

esp-idf-monitor will read settings from other usual configuration files if no other configuration file is used. It automatically reads from setup.cfg or tox.ini if they exist.

Configuration Options  Below is a table listing the available configuration options:
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>menu_key</td>
<td>Key to access the main menu.</td>
<td>T</td>
</tr>
<tr>
<td>exit_key</td>
<td>Key to exit the monitor.</td>
<td>]</td>
</tr>
<tr>
<td>chip_reset_key</td>
<td>Key to initiate a chip reset.</td>
<td>R</td>
</tr>
<tr>
<td>recompile_upload_key</td>
<td>Key to recompile and upload.</td>
<td>F</td>
</tr>
<tr>
<td>recompile_upload_app_key</td>
<td>Key to recompile and upload just the application.</td>
<td>A</td>
</tr>
<tr>
<td>toggle_output_key</td>
<td>Key to toggle the output display.</td>
<td>Y</td>
</tr>
<tr>
<td>toggle_log_key</td>
<td>Key to toggle the logging feature.</td>
<td>L</td>
</tr>
<tr>
<td>toggle_timestamp_key</td>
<td>Key to toggle timestamp display.</td>
<td>I</td>
</tr>
<tr>
<td>chip_reset_bootloader_key</td>
<td>Key to reset the chip to bootloader mode.</td>
<td>P</td>
</tr>
<tr>
<td>exit_menu_key</td>
<td>Key to exit the monitor from the menu.</td>
<td>X</td>
</tr>
<tr>
<td>skip_menu_key</td>
<td>Pressing the menu key can be skipped for menu commands.</td>
<td>False</td>
</tr>
</tbody>
</table>

**Syntax**  The configuration file is in .ini file format: it must be introduced by an `[esp-idf-monitor]` header to be recognized as valid. This section then contains name = value entries. Lines beginning with # or ; are ignored as comments.

```ini
# esp-idf-monitor.cfg file to configure internal settings of esp-idf-monitor
[esp-idf-monitor]
menu_key = T
exit_key = ]
chip_reset_key = R
recompile_upload_key = F
recompile_upload_app_key = A
toggle_output_key = Y
toggle_log_key = L
toggle_timestamp_key = I
chip_reset_bootloader_key = P
exit_menu_key = X
skip_menu_key = False
```

**Known Issues with IDF Monitor**

**Issues Observed on Windows**

- Arrow keys, as well as some other keys, do not work in GDB due to Windows Console limitations.
- Occasionally, when `idf.py` 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.

**Standard Toolchain Setup for Linux and macOS**

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

**Setting up Development Environment**  These are the steps for setting up the ESP-IDF for your ESP32-S2.

- **Step 1. Install Prerequisites**
- **Step 2. Get ESP-IDF**
- **Step 3. Set up the Tools**
- **Step 4. Set up the Environment Variables**
- **Step 5. First Steps on ESP-IDF**
Step 1. Install Prerequisites  In order to use ESP-IDF with the ESP32-S2, you need to install some software packages based on your Operating System. This setup guide helps you on getting everything installed on Linux and macOS based systems.

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

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

- CentOS 7 & 8:
  ```
  sudo yum -y update && sudo yum install git wget flex bison gperf python3 cmake ninja-build ccache dfu-util libusb
  ```

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

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

Note:

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

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

- Install CMake & Ninja build:
  
  - If you have HomeBrew, you can run:
    ```
    brew install cmake ninja dfu-util
    ```
  
  - If you have MacPorts, you can run:
    ```
    sudo port install cmake ninja dfu-util
    ```
  
  - 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:

```bash
```

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

Apple M1 Users  If you use Apple M1 platform and see an error like this:
Chapter 1. Get Started

### WARNING: directory for tool xtensa-esp32-elf version esp-2021r2-patch3-8.4.0 is present, but tool was not found

**ERROR:** tool xtensa-esp32-elf has no installed versions. Please run ‘install.sh’ to install it.

or:

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

Then you need to install Apple Rosetta 2 by running

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

### Installing Python 3

Based on macOS [Catalina 10.15 release notes](https://developer.apple.com/documentation/developer-tools/catalina), use of Python 2.7 is not recommended and Python 2.7 is not 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 is not already installed on your computer:

```
python3 --version
```

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

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

- Installing with [HomeBrew](https://brew.sh/) can be done as follows:

  ```
  brew install python3
  ```

- If you have [MacPorts](https://port供), you can run:

  ```
  sudo port install python38
  ```

### Step 2. Get ESP-IDF

To build applications for the ESP32-S2, 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.

Open Terminal, and run the following commands:

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

ESP-IDF is downloaded into `~/esp/esp-idf`.

Consult [ESP-IDF Versions](https://github.com/espressif/esp-idf/releases) for information about which ESP-IDF version to use in a given situation.

### 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, for projects supporting ESP32-S2.

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

or with Fish shell

```
cd ~/esp/esp-idf
/install.sh esp32s2
```
Chapter 1. Get Started

```bash
cd ~/esp/esp-idf
./install.fish esp32s2
```

The above commands install tools for ESP32-S2 only. If you intend to develop projects for more chip targets then you should list all of them and run for example:

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

or with Fish shell

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

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

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

or with Fish shell

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

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

```bash
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: unable...
```

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

**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 does not change the URLs used to access any Git repositories.

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

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

**Note:** For users in China, we recommend using our download server located in China for faster download speed.

```bash
cd ~/esp/esp-idf
export IDF_GITHUB_ASSETS="dl.espressif.cn/github_assets"
./install.sh
```
Chapter 1. Get Started

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. 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 account 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.

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.

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

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

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

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

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.

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

This guide helps you on the first steps using ESP-IDF. Follow this guide to start a new project on the ESP32-S2 and build, flash, and monitor the device output.

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

Start a Project   Now you are ready to prepare your application for ESP32-S2. You can start with get-started/hello_world project from examples directory in ESP-IDF.

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

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

```
cd ~/esp
cp -r $IDF_PATH/examples/get-started/hello_world .
```
Chapter 1. Get Started

**Note:** 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.

**Connect Your Device**  Now connect your ESP32-S2 board to the computer and check under which serial port the board is visible.

Serial ports have the following naming patterns:

- **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-S2* for full details.

**Note:** Keep the port name handy as it is needed in the next steps.

**Configure Your Project**  Navigate to your `hello_world` directory, set ESP32-S2 as the target, and run the project configuration utility `menuconfig`.

```
$ cd ~/esp/hello_world
$ idf.py set-target esp32s2
$ idf.py menuconfig
```

After opening a new project, you should first set the target with `idf.py set-target esp32s2`. Note that existing builds and configurations in the project, if any, are cleared and initialized in this process. The target may be saved in the environment variable to skip this step at all. See *Select the Target Chip: set-target* for additional information.

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

![Project configuration - Home window](image)

**Fig. 57:** Project configuration - Home window

You are using this menu to set up project specific variables, e.g., Wi-Fi network name and password, the processor speed, etc. Setting up the project with menuconfig may be skipped for “hello_world”, since this example runs with default configuration.
Chapter 1. Get Started

**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.

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

**Console Output Configuration** If you are using the USB for flashing the ESP32-S2, you need to change the channel for the console output from UART (default) to USB.

1. Navigate to the option `Channel for console output`.
2. Change to the option (the default is always UART):
   - USB CDC
3. Save the new configuration and exit the `menuconfig` screen.

**Build the Project** Build the project by running:

```
idf.py build
```

This command compiles the application and all ESP-IDF components, then it generates 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
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 finishes by generating the firmware binary .bin files.

**Flash onto the Device** To flash the binaries that you just built for the ESP32-S2 in the previous step, you need to run the following command:

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

Replace `PORT` with your ESP32-S2 board’s USB port name. If the `PORT` is not defined, the `idf.py` will try to connect automatically using the available USB ports.

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.
Encountered Issues While Flashing? See the “Additional Tips” below. You can also refer to Flashing Troubleshooting page or Establish Serial Connection with ESP32-S2 for more detailed information.

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

```
... 
esptool.py --chip esp32s2 -p /dev/ttyUSB0 -b 460800 --before=default_reset -- 
→ after=hard_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB... 
→ 0x8000 partition_table/partition-table.bin 0x1000 bootloader/bootloader.bin... 
→ 0x10000 hello_world.bin 
esptool.py v3.0-dev 
Serial port /dev/ttyUSB0 
Connecting.... 
Chip is ESP32-S2 
Features: WiFi 
Crystal is 40MHz 
Uploading stub... 
Running stub... 
Stub running... 
Changing baud rate to 460800 
Changed. 
Configuring flash size... 
Compressed 3072 bytes to 103... 
Writing at 0x00008000 in 0.0 seconds (effective 3851.6 kbit/s)... 
Hash of data verified. 
Compressed 22592 bytes to 13483... 
Writing at 0x00001000 in 0.3 seconds (effective 595.1 kbit/s)... 
Hash of data verified. 
Compressed 140048 bytes to 70298... 
Writing at 0x00010000 (20 %) 
Writing at 0x00014000 (40 %) 
Writing at 0x00018000 (60 %) 
Writing at 0x0001c000 (80 %) 
Writing at 0x00020000 (100 %) 
Wrote 140048 bytes (70298 compressed) at 0x00010000 in 1.7 seconds (effective 662.5 kbit/s)... 
Hash of data verified. 
Leaving... 
Hard resetting via RTS pin... 
Done
```

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

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

**Monitor the Output** 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 <PORT> 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"...
```
After startup and diagnostic logs scroll up, you should see “Hello world!” printed out by the application.

...  
Hello world!  
Restarting in 10 seconds...  
This is esp32s2 chip with 1 CPU core(s), WiFi, silicon revision 0, 2 MB...  
→ external flash  
Minimum free heap size: 253900 bytes  
Restarting in 9 seconds...  
Restarting in 8 seconds...  
Restarting in 7 seconds...

To exit IDF monitor use the shortcut Ctrl+].

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 is all that you need to get started with ESP32-S2!

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

Important: Some of examples do not support ESP32-S2 because required hardware is not included in ESP32-S2 so it cannot be supported.

If building an example, please check the README file for the **Supported Targets** table. If this is present including ESP32-S2 target, or the table does not exist at all, the example will work on ESP32-S2.

Additional Tips

**Permission Denied Issue**  With some Linux distributions, you may get the error message similar to *Could not open port <PORT>: Permission denied: '<PORT>'* when flashing the ESP32-S2. *This can be solved by adding the current user to the specific group*, such as dialout or uucp group.

**Python Compatibility**  ESP-IDF supports Python 3.8 or newer. It is recommended to upgrade your operating system to a recent version satisfying this requirement. Other options include the installation of Python from **sources** or the use of a Python version management system such as **pyenv**.

**Start with Board Support Package**  To speed up prototyping on some development boards, you can use **Board Support Packages (BSPs)**, which makes initialization of a particular board as easy as few function calls.
A BSP typically supports all of the hardware components provided on development board. Apart from the pinout definition and initialization functions, a BSP ships with drivers for the external components such as sensors, displays, audio codecs etc.

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

Here is an example of how to add ESP32-S2-Kaluga-Kit BSP to your project:

```
idf.py add-dependency esp32_s2_kaluga_kit
```

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

**Flash Erase** Erasing the flash is also possible. To erase the entire flash memory you can run the following command:

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

For erasing the OTA data, if present, you can run this command:

```
idf.py -p PORT erase-otadata
```

The flash erase command can take a while to be done. Do not disconnect your device while the flash erasing is in progress.

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

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.

**Related Documents**

- Establish Serial Connection with ESP32-S2
- Eclipse Plugin
- VSCode Extension
- IDF Monitor

1.4 Build Your First Project

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

1.5 Uninstall ESP-IDF

If you want to remove ESP-IDF, please follow Uninstall ESP-IDF.
Chapter 2

API Reference

2.1 API Conventions

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

ESP-IDF provides several kinds of programming interfaces:

- C functions, structures, enums, type definitions, and preprocessor macros declared in public header files of ESP-IDF components. Various pages in the API Reference section of the programming guide contain descriptions of these functions, structures, and types.
- Build system functions, predefined variables, and options. These are documented in the ESP-IDF CMake Build System API.
- Kconfig options can be used in code and in the build system (CMakeLists.txt) files.
- Host tools and their command line parameters are also part of the ESP-IDF interfaces.

ESP-IDF is made up of multiple components where these components either contain code specifically written for ESP chips, or contain a third-party library (i.e., a third-party component). In some cases, third-party components contain an “ESP-IDF specific” wrapper in order to provide an interface that is either simpler or better integrated with the rest of ESP-IDF’s features. In other cases, third-party components present the original API of the underlying library directly.

The following sections explain some of the aspects of ESP-IDF APIs and their usage.

2.1.1 Error Handling

Most ESP-IDF APIs return error codes defined with the `esp_err_t` type. See Error Handling section for more information about error handling approaches. Error Codes Reference contains the list of error codes returned by ESP-IDF components.

2.1.2 Configuration Structures

**Important:** Correct initialization of configuration structures is an important part of making the application compatible with future versions of ESP-IDF.
Most initialization, configuration, and installation functions in ESP-IDF (typically named ..._init(), ..._config(), and ..._install()) take a configuration structure pointer as an argument. For example:

```c
const esp_timer_create_args_t my_timer_args = {
  .callback = &my_timer_callback,
  .arg = callback_arg,
  .name = "my_timer"
};
esp_timer_handle_t my_timer;
esp_err_t err = esp_timer_create(&my_timer_args, &my_timer);
```

These functions never store the pointer to the configuration structure, so it is safe to allocate the structure on the stack. The application must initialize all fields of the structure. The following is incorrect:

```c
esp_timer_create_args_t my_timer_args;
my_timer_args.callback = &my_timer_callback;
/* Incorrect! Fields .arg and .name are not initialized */
esp_timer_create(&my_timer_args, &my_timer);
```

Most ESP-IDF examples use C99 designated initializers for structure initialization since they provide a concise way of setting a subset of fields, and zero-initializing the remaining fields:

```c
const esp_timer_create_args_t my_timer_args = {
  .callback = &my_timer_callback,
  /* Correct, fields .arg and .name are zero-initialized */
};
```

The C++ language supports designated initializer syntax, too, but the initializers must be in the order of declaration. When using ESP-IDF APIs in C++ code, you may consider using the following pattern:

```c
/* Correct, fields .dispatch_method, .name and .skip_unhandled_events are zero-
   →initialized */
const esp_timer_create_args_t my_timer_args = {
  .callback = &my_timer_callback,
  /* Correct, fields .arg and .name are zero-initialized */
};
```

For more information on designated initializers, see Designated Initializers. Note that C++ language versions older than C++20, which are not the default in the current version of ESP-IDF, do not support designated initializers. If you have to compile code with an older C++ standard than C++20, you may use GCC extensions to produce the following pattern:

```c
esp_timer_create_args_t my_timer_args = {};
/* All the fields are zero-initialized */
my_timer_args.callback = &my_timer_callback;
```

### Default Initializers

For some configuration structures, ESP-IDF provides macros for setting default values of fields:

```c
httpd_config_t config = HTTPD_DEFAULT_CONFIG();
/* HTTPD_DEFAULT_CONFIG expands to a designated initializer. Now all fields are-
   →set to the default values, and any field can still be modified: */
config.server_port = 8081;
```

(continues on next page)
httpd_handle_t server;
esp_err_t err = httpd_start(&server, &config);

It is recommended to use default initializer macros whenever they are provided for a particular configuration structure.

2.1.3 Private APIs

Certain header files in ESP-IDF contain APIs intended to be used only in ESP-IDF source code rather than by
the applications. Such header files often contain private or esp_private in their name or path. Certain
components, such as hal only contain private APIs.

Private APIs may be removed or changed in an incompatible way between minor or patch releases.

2.1.4 Components in Example Projects

ESP-IDF examples contain a variety of projects demonstrating the usage of ESP-IDF APIs. In order to reduce code
duplication in the examples, a few common helpers are defined inside components that are used by multiple examples.
This includes components located in common_components directory, as well as some of the components located in
the examples themselves. These components are not considered to be part of the ESP-IDF API.

It is not recommended to reference these components directly in custom projects (via EXTRA_COMPONENT_DIRS
build system variable), as they may change significantly between ESP-IDF versions. When starting a new project
based on an ESP-IDF example, copy both the project and the common components it depends on out of ESP-IDF,
and treat the common components as part of the project. Note that the common components are written with examples
in mind, and might not include all the error handling required for production applications. Before using, take time to
read the code and understand if it is applicable to your use case.

2.1.5 API Stability

ESP-IDF uses Semantic Versioning as explained in the Versioning Scheme.

Minor and bugfix releases of ESP-IDF guarantee compatibility with previous releases. The sections below explain
different aspects and limitations to compatibility.

Source-level Compatibility

ESP-IDF guarantees source-level compatibility of C functions, structures, enums, type definitions, and preprocessor
macros declared in public header files of ESP-IDF components. Source-level compatibility implies that the application
source code can be recompiled with the newer version of ESP-IDF without changes.

The following changes are allowed between minor versions and do not break source-level compatibility:

• Deprecating functions (using the deprecated attribute) and header files (using a preprocessor #warning).
  Deprecations are listed in ESP-IDF release notes. It is recommended to update the source code to use the newer
  functions or files that replace the deprecated ones, however, this is not mandatory. Deprecated functions and
  files can be removed from major versions of ESP-IDF.
• Renaming components, moving source and header files between components  ——provided that the build system
  ensures that correct files are still found.
• Renaming Kconfig options. Kconfig system’s backward compatibility ensures that the original Kconfig option
  names can still be used by the application in sdkconfig file, CMake files, and source code.
Lack of Binary Compatibility

ESP-IDF does not guarantee binary compatibility between releases. This means that if a precompiled library is built with one ESP-IDF version, it is not guaranteed to work the same way with the next minor or bugfix release. The following are the possible changes that keep source-level compatibility but not binary compatibility:

- Changing numerical values for C enum members.
- Adding new structure members or changing the order of members. See Configuration Structures for tips that help ensure compatibility.
- Replacing an `extern` function with a `static inline` one with the same signature, or vice versa.
- Replacing a function-like macro with a compatible C function.

Other Exceptions from Compatibility

While we try to make upgrading to a new ESP-IDF version easy, there are parts of ESP-IDF that may change between minor versions in an incompatible way. We appreciate issuing reports about any unintended breaking changes that do not fall into the categories below.

- Private APIs.
- Components in Example Projects.
- Features clearly marked as “beta”, “preview”, or “experimental”.
- Changes made to mitigate security issues or to replace insecure default behaviors with secure ones.
- Features that were never functional. For example, if it was never possible to use a certain function or an enumeration value, it may get renamed (as part of fixing it) or removed. This includes software features that depend on non-functional chip hardware features.
- Unexpected or undefined behavior that is not documented explicitly may be fixed/changed, such as due to missing validation of argument ranges.
- Location of `Kconfig` options in menuconfig.
- Location and names of example projects.

2.2 Application Protocols

2.2.1 ASIO Port

ASIO is a cross-platform C++ library, see https://think-async.com/Asio/. It provides a consistent asynchronous model using a modern C++ approach.

The ESP-IDF component ASIO has been moved from ESP-IDF since version v5.0 to a separate repository:

- ASIO component on GitHub

To add ASIO component in your project, please run `idf.py add-dependency espressif/asio`.

Hosted Documentation

The documentation can be found on the link below:

- ASIO documentation (English)
2.2.2 ESP-Modbus

The Espressif ESP-Modbus Library (esp-modbus) supports Modbus communication in the networks based on RS485, Wi-Fi, and Ethernet interfaces. Since ESP-IDF version v5.0, the component freemodbus has been moved from ESP-IDF to a separate repository:

- ESP-Modbus component on GitHub

Hosted Documentation

The documentation can be found through the link below:

- ESP-Modbus documentation (English)

Application Example

The examples below demonstrate the ESP-Modbus library of serial and TCP ports for both slave and master implementations respectively.

- protocols/modbus/serial/mb_slave
- protocols/modbus/serial/mb_master
- protocols/modbus/tcp/mb_tcp_slave
- protocols/modbus/tcp/mb_tcp_master

Please refer to the README.md documents of each specific example for details.

Protocol References

- For the detailed protocol specifications, see The Modbus Organization.

2.2.3 ESP-MQTT

Overview

ESP-MQTT is an implementation of MQTT protocol client, which is a lightweight publish/subscribe messaging protocol. Now ESP-MQTT supports MQTT v5.0.

Features

- Support MQTT over TCP, SSL with Mbed TLS, MQTT over WebSocket, and 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 Quality of Service (QoS) levels (it should be a fully functional client)

Application Examples

- protocols/mqtt/tcp: MQTT over TCP, default port 1883
- protocols/mqtt/ssl: MQTT over TLS, default port 8883
- protocols/mqtt/ssl_ds: MQTT over TLS using digital signature peripheral for authentication, default port 8883
- protocols/mqtt/ssl Mutual Auth: MQTT over TLS using certificates for authentication, default port 8883
• protocols/mqtt/ssl_psk: MQTT over TLS 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
• protocols/mqtt5: Uses ESP-MQTT library to connect to broker with MQTT v5.0

MQTT Message Retransmission

A new MQTT message is created by calling `esp_mqtt_client_publish` or its non blocking counterpart `esp_mqtt_client_enqueue`.

Messages with QoS 0 is sent only once. QoS 1 and 2 have different behaviors since the protocol requires extra steps to complete the process.

The ESP-MQTT library opts to always retransmit unacknowledged QoS 1 and 2 publish messages to avoid losses in faulty connections, even though the MQTT specification requires the re-transmission only on reconnect with Clean Session flag been set to 0 (set `disable_clean_session` to true for this behavior).

QoS 1 and 2 messages that may need retransmission are always enqueued, but first transmission try occurs immediately if `esp_mqtt_client_publish` is used. A transmission retry for unacknowledged messages will occur after `message_retransmit_timeout`. After `CONFIG_MQTT_OUTBOX_EXPIRED_TIMEOUT_MS` messages will expire and be deleted. If `CONFIG_MQTT_REPORT_DELETED_MESSAGES` is set, an event will be sent to notify the user.

Configuration

The configuration is made by setting fields in `esp_mqtt_client_config_t` struct. The configuration struct has the following sub structs to configure different aspects of the client operation.

• `esp_mqtt_client_config_t::broker_t` - Allow to set address and security verification.
• `esp_mqtt_client_config_t::credentials_t` - Client credentials for authentication.
• `esp_mqtt_client_config_t::session_t` - Configuration for MQTT session aspects.
• `esp_mqtt_client_config_t::network_t` - Networking related configuration.
• `esp_mqtt_client_config_t::task_t` - Allow to configure FreeRTOS task.
• `esp_mqtt_client_config_t::buffer_t` - Buffer size for input and output.

In the following sections, the most common aspects are detailed.

Broker

Address  Broker address can be set by usage of `address` struct. The configuration can be made by usage of `uri` field or the combination of `hostname, transport` and `port`. Optionally, `path` could be set, this field is useful in WebSocket connections.

The `uri` field is used in the format `scheme://hostname:port/path`.

• Currently support `mqtt, mqtts, ws, wss` schemes
• MQTT over TCP samples:
  - `mqtt://mqtt.eclipseprojects.io`: MQTT over TCP, default port 1883
  - `mqtt://mqtt.eclipseprojects.io:1884`: MQTT over TCP, port 1884
  - `mqtt://username:password@mqtt.eclipseprojects.io:1884`: MQTT over TCP, port 1884, with username and password
• MQTT over SSL samples:
  - `mqtts://mqtt.eclipseprojects.io`: MQTT over SSL, port 8883
  - `mqtts://mqtt.eclipseprojects.io:8884`: MQTT over SSL, port 8884
• MQTT over WebSocket samples:
  - `ws://mqtt.eclipseprojects.io:80/mqtt`
• MQTT over WebSocket Secure samples:
  - `wss://mqtt.eclipseprojects.io:443/mqtt`
• Minimal configurations:
const esp_mqtt_client_config_t mqtt_cfg = {
    .broker.address.uri = "mqtt://mqtt.eclipseprojects.io",
};
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.).

Verification For secure connections with TLS used, and to guarantee Broker’s identity, the verification struct must be set. The broker certificate may be set in PEM or DER format. To select DER, the equivalent certificate_len field must be set. Otherwise, a null-terminated string in PEM format should be provided to certificate field.

- Get certificate from server, example: mqtt.eclipseprojects.io
  openssl s_client -showcerts -connect mqtt.eclipseprojects.io:8883 < /dev/null \
  2> /dev/null | openssl x509 -outform PEM > mqtt_eclipse_org.pem
- Check the sample application: protocols/mqtt/ssl
- Configuration:

Get certificate from server, example: mqtt.eclipseprojects.io

const esp_mqtt_client_config_t mqtt_cfg = {
    .broker = {
        .address.uri = "mqtts://mqtt.eclipseprojects.io:8883",
        .verification.certificate = (const char*)mqtt_eclipse_org_pem_start,
    },
};

For details about other fields, please check the API Reference and TLS Server Verification.

Client Credentials All client related credentials are under the credentials field.

- username: pointer to the username used for connecting to the broker, can also be set by URI
- client_id: pointer to the client ID, defaults to ESP32_%CHIPID% where %CHIPID% are the last 3 bytes of MAC address in hex format

Authentication It is possible to set authentication parameters through the authentication field. The client supports the following authentication methods:

- password: use a password by setting
- certificate and key: mutual authentication with TLS, and both can be provided in PEM or DER format
- use_secure_element: use secure element available in ESP32-WROOM-32SE
- ds_data: use Digital Signature Peripheral available in some Espressif devices

Session For MQTT session related configurations, session fields should be used.

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 last_will struct.

- topic: pointer to the LWT message topic
- msg: pointer to the LWT message
- msg_len: length of the LWT message, required if msg is not null-terminated
Chapter 2. API Reference

- **qos**: quality of service for the LWT message
- **retain**: specifies the retain flag of the LWT message

**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**: enable 3.1.1 version of MQTT protocol
- **CONFIG_MQTT_TRANSPORT_SSL** and **CONFIG_MQTT_TRANSPORT_WEBSOCKET**: enable specific MQTT transport layer, such as SSL, WEBSOCKET, and WEBSOCKET_SECURE
- **CONFIG_MQTT_CUSTOM_OUTBOX**: disable 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 contains the message ID of the subscribe message.
- **MQTT_EVENT_UNSUBSCRIBED**: The broker has acknowledged the client’s unsubscribe request. The event data contains the message ID of the unsubscribe message.
- **MQTT_EVENT_PUBLISHED**: The broker has acknowledged the client’s publish message. This is only posted for QoS level 1 and 2, as level 0 does not use acknowledgements. The event data contains 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 events are 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. The field `error_handle` in the event data contains `error_type` that can be used to identify the error. The type of error determines which parts of the `error_handle` struct is filled.

**API Reference**

**Header File**

- **components/mqtt/esp-mqtt/include/mqtt_client.h**
  
  This header file can be included with:

  ```
  #include "mqtt_client.h"
  ```

  This header file is a part of the API provided by the mqtt component. To declare that your component depends on mqtt, add the following to your CMakeLists.txt:

  ```
  REQUIRES mqtt
  ```
  or

  ```
  PRIV_REQUIRES mqtt
  ```

**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.

**Parameters**
- config: `MQTT` configuration structure

**Returns**
- `mqCONN_client_handle` if successfully created, NULL on error

**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.

**Parameters**
- client: `MQTT` client handle
- uri: URI

**Returns**
- ESP_FAIL if URI parse error, ESP_OK on success

**esp_err_t esp_mqtt_client_start** (esp_mqtt_client_handle_t client)

Starts `MQTT` client with already created client handle.

**Parameters**
- client: `MQTT` client handle

**Returns**
- ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization ESP_FAIL on other error

**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.

**Parameters**
- client: `MQTT` client handle

**Returns**
- ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization ESP_FAIL if client is in invalid state

**esp_err_t esp_mqtt_client_disconnect** (esp_mqtt_client_handle_t client)

This api is typically used to force disconnection from the broker.

**Parameters**
- client: `MQTT` client handle

**Returns**
- ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization

**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

**Parameters**
- client: `MQTT` client handle

**Returns**
- ESP_OK on success ESP_ERR_INVALID_ARG on wrong initialization ESP_FAIL if client is in invalid state

**int esp_mqtt_client_subscribe_single** (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.
- esp_mqtt_client_subscribe could be used to call this function.

**Parameters**
- client: `MQTT` client handle
- topic: topic filter to subscribe
- qos: Max qos level of the subscription

**Returns**
- message_id of the subscribe message on success -1 on failure -2 in case of full outbox.
int esp_mqtt_client_subscribe_multiple (esp_mqtt_client_handle_t client, const esp_mqtt_topic_t *topic_list, int size)

Subscribe the client to a list of defined topics 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.
• esp_mqtt_client_subscribe could be used to call this function.

Parameters
• client -MQTT client handle
• topic_list -List of topics to subscribe
• size - size of topic_list

Returns message_id of the subscribe message on success -1 on failure -2 in case of full outbox.

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_single for details

Parameters
• client -MQTT client handle
• topic

Returns message_id of the subscribe message on success -1 on failure

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

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

Returns message_id of the publish message (for QoS 0 message_id will always be zero) on success.
-1 on failure, -2 in case of full outbox.

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().

**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 QoS 1 and QoS 2 are enqueued

**Returns**
- message_id if queued successfully, -1 on failure, -2 in case of full outbox.

```c
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

**Parameters**
- `client` – MQTT client handle

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG on wrong initialization

```c
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.

**Parameters**
- `client` – MQTT client handle
- `config` – MQTT configuration structure

**Returns**
- ESP_ERR_NO_MEM if failed to allocate
- ESP_ERR_INVALID_ARG if conflicts on transport configuration.
- ESP_OK on success

```c
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.

**Parameters**
- `client` – MQTT client handle
- `event` – event type
- `event_handler` – handler callback
- `event_handler_arg` – handlers context

**Returns**
- ESP_ERR_NO_MEM if failed to allocate
- ESP_ERR_INVALID_ARG if conflicts on transport configuration.
- ESP_OK on success

```c
esp_err_t esp_mqtt_client_unregister_event(esp_mqtt_client_handle_t client, esp_mqtt_event_id_t event, esp_event_handler_t event_handler)
```

Unregisters mqtt event.

**Parameters**
- `client` – mqtt client handle
- `event` – event ID
- `event_handler` – handler to unregister

**Returns**
- ESP_ERR_NO_MEM if failed to allocate
- ESP_ERR_INVALID_ARG on invalid event ID
- ESP_OK on success

```c
int esp_mqtt_client_get_outbox_size(esp_mqtt_client_handle_t client)
```

Get outbox size.

**Parameters**
- `client` – MQTT client handle
### Returns
outbox size 0 on wrong initialization

```c
esp_err_t esp_mqtt_dispatch_custom_event (esp_mqtt_client_handle_t client, esp_mqtt_event_t *event)
```

Dispatch user event to the mqtt internal event loop.

### Parameters
- `client` - `MQTT` client handle
- `event` - `MQTT` event handle structure

### Returns
ESP_OK on success ESP_ERR_TIMEOUT if the event couldn’t be queued (ref also CONFIG_MQTT_EVENT_QUEUE_SIZE)

### 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:

<table>
<thead>
<tr>
<th>error_type</th>
<th>related member variables</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQTT_ERROR_TYPE_TCP_TRANSPORT</td>
<td>esp_tls_last_err, esp_tls_stack_err, esp_tls_cert_verify_flags, sock_errno</td>
<td>Error reported from tcp_transport/esp-tls</td>
</tr>
<tr>
<td>MQTT_ERROR_TYPE_CONNECTION_REFUSED</td>
<td>connect_return_code</td>
<td>Internal error reported from MQTT broker on connection</td>
</tr>
</tbody>
</table>

### Public Members

**esp_err_t esp_tls_last_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

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* message 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

bool `retain`

Retained flag of the message associated with this event

int `qos`

QoS of the messages associated with this event

bool `dup`

dup flag of the message associated with this event

**esp_mqtt_protocol_ver_t** `protocol_ver`

MQTT protocol version used for connection, defaults to value from menuconfig

struct **esp_mqtt_client_config_t**

*MQTT* client configuration structure
• Default values can be set via menuconfig
• All certificates and key data could be passed in PEM or DER format. PEM format must have a terminating NULL character and the related len field set to 0. DER format requires a related len field set to the correct length.

**Public Members**

```c
struct esp_mqtt_client_config_t::broker_t broker
    Broker address and security verification

struct esp_mqtt_client_config_t::credentials_t credentials
    User credentials for broker

struct esp_mqtt_client_config_t::session_t session
    MQTT session configuration.

struct esp_mqtt_client_config_t::network_t network
    Network configuration

struct esp_mqtt_client_config_t::task_t task
    FreeRTOS task configuration.

struct esp_mqtt_client_config_t::buffer_t buffer
    Buffer size configuration.

struct esp_mqtt_client_config_t::outbox_config_t outbox
    Outbox configuration.

struct broker_t
    Broker related configuration

Public Members

```c
struct esp_mqtt_client_config_t::broker_t::address_t address
    Broker address configuration

struct esp_mqtt_client_config_t::broker_t::verification_t verification
    Security verification of the broker

struct address_t
    Broker address

• uri have precedence over other fields
• If uri isn’t set at least hostname, transport and port should.
Public Members

const char *uri
  Complete MQTT broker URI

const char *hostname
  Hostname, to set ipv4 pass it as string

`esp_mqtt_transport_t` transport
  Selects transport

const char *path
  Path in the URI

uint32_t port
  MQTT server port

struct verification_t
  Broker identity verification
  If fields are not set broker’s identity isn’t verified. it’s recommended to set the options in this
  struct for security reasons.

Public Members

bool use_global_ca_store
  Use a global ca_store, look esp-tls documentation for details.

`esp_err_t` (*`crt_bundle_attach`)(void *conf)
  Pointer to ESP x509 Certificate Bundle attach function for the usage of certificate bundles.

const char *certificate
  Certificate data, default is NULL, not required to verify the server.

size_t certificate_len
  Length of the buffer pointed to by certificate.

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). PSK is enabled only if there are no other ways to verify broker.

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

const char **alpn_protos
  NULL-terminated list of supported application protocols to be used for ALPN
const char *\texttt{common\_name}

Pointer to the string containing server certificate common name. If non-NULL, server certificate CN must match this name. If NULL, server certificate CN must match hostname. This is ignored if \texttt{skip\_cert\_common\_name\_check=true}.

\begin{verbatim}
struct buffer_t
    Client buffer size configuration
    Client have two buffers for input and output respectively.
\end{verbatim}

**Public Members**

\begin{verbatim}
int size
    size of MQTT send/receive buffer

int out_size
    size of MQTT output buffer. If not defined, defaults to the size defined by buffer_size
\end{verbatim}

\begin{verbatim}
struct credentials_t
    Client related credentials for authentication.
\end{verbatim}

**Public Members**

\begin{verbatim}
const char *\texttt{username}
    MQTT username

const char *\texttt{client\_id}
    Set MQTT client identifier. Ignored if \texttt{set\_null\_client\_id == true}. If NULL set the default client id. Default client id is ESP32\_CHIPID\% where CHIPID\% are last 3 bytes of MAC address in hex format

bool set_null_client_id
    Selects a NULL client id
\end{verbatim}

\begin{verbatim}
struct esp_mqtt_client_config_t::credentials_t::authentication_t authentication
    Client authentication
\end{verbatim}

\begin{verbatim}
struct authentication_t
    Client authentication
    Fields related to client authentication by broker
    For mutual authentication using TLS, user could select certificate and key, secure element or digital signature peripheral if available.
\end{verbatim}

**Public Members**

\begin{verbatim}
const char *\texttt{password}
    MQTT password
\end{verbatim}
const char *certificate
Certificate for ssl mutual authentication, not required if mutual authentication is not needed.
Must be provided with key.

size_t certificate_len
Length of the buffer pointed to by certificate.

const char *key
Private key for SSL mutual authentication, not required if mutual authentication is not needed.
If it is not NULL, also certificate has to be provided.

size_t key_len
Length of the buffer pointed to by key.

const char *key_password
Client key decryption password, not PEM nor DER, if provided key_password_len must be correctly set.

int key_password_len
Length of the password pointed to by key_password

bool use_secure_element
Enable secure element, available in ESP32-ROOM-32SE, for SSL connection

void *ds_data
Carrier of handle for digital signature parameters, digital signature peripheral is available in some Espressif devices.

struct network_t
Network related configuration

Public Members

int reconnect_timeout_ms
Reconnect to the broker after this value in miliseconds if auto reconnect is not disabled (defaults to 10s)

int timeout_ms
Abort network operation if it is not completed after this value, in milliseconds (defaults to 10s).

int refresh_connection_after_ms
Refresh connection after this value (in milliseconds)

bool disable_auto_reconnect
Client will reconnect to server (when errors/disconnect). Set disable_auto_reconnect=true to disable
esp_transport_handle_t transport

Custom transport handle to use. Warning: The transport should be valid during the client lifetime and is destroyed when esp_mqtt_client_destroy is called.

struct ifreq *if_name

The name of interface for data to go through. Use the default interface without setting

struct outbox_config_t

Client outbox configuration options.

**Public Members**

uint64_t limit

Size limit for the outbox in bytes.

struct session_t

MQTT Session related configuration

**Public Members**

struct esp_mqtt_client_config_t::session_t::last_will_t last_will

Last will configuration

bool disable_clean_session

MQTT clean session, default clean_session is true

int keepalive

MQTT keepalive, default is 120 seconds When configuring this value, keep in mind that the client attempts to communicate with the broker at half the interval that is actually set. This conservative approach allows for more attempts before the broker’s timeout occurs

bool disable_keepalive

Set disable_keepalive=true to turn off keep-alive mechanism, keepalive is active by default. Note: setting the config value keepalive to 0 doesn’t disable keepalive feature, but uses a default keepalive period

esp_mqtt_protocol_ver_t protocol_ver

MQTT protocol version used for connection.

int message_retransmit_timeout

timeout for retransmitting of failed packet

struct last_will_t

Last Will and Testament message configuration.
Chapter 2. API Reference

Public Members

const char *topic
LWT (Last Will and Testament) message topic

const char *msg
LWT message, may be NULL terminated

int msg_len
LWT message length, if msg isn’t NULL terminated must have the correct length

int qos
LWT message QoS

int retain
LWT retained message flag

struct task_t
Client task configuration

Public Members

int priority
MQTT task priority

int stack_size
MQTT task stack size

struct topic_t
Topic definition struct

Public Members

const char *filter
Topic filter to subscribe

int qos
Max QoS level of the subscription

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
**esp_mqtt_client_subscribe** (client_handle, topic_type, qos_or_size)

Convenience macro to select subscribe function to use.

Notes:

- Usage of `esp_mqtt_client_subscribe_single` is the same as previous `esp_mqtt_client_subscribe`, refer to it for details.

### Parameters

- **client_handle** - MQTT client handle
- **topic_type** - Needs to be char* for single subscription or `esp_mqtt_topic_t` for multiple topics
- **qos_or_size** - It’s either a qos when subscribing to a single topic or the size of the subscription array when subscribing to multiple topics.

### Returns

Message_id of the subscribe message on success -1 on failure -2 in case of full outbox.

## Type Definitions

typedef struct esp_mqtt_client * **esp_mqtt_client_handle_t**

typedef enum **esp_mqtt_event_id_t**

- *MQTT* event types.

  - User event handler receives context data in `esp_mqtt_event_t` structure with
    - client - MQTT client handle
    - various other data depending on event type

  typedef enum **esp_mqtt_connect_return_code_t**

  - *MQTT* connection error codes propagated via ERROR event

  typedef enum **esp_mqtt_error_type_t**

  - *MQTT* connection error codes propagated via ERROR event

  typedef enum **esp_mqtt_transport_t**

  - *MQTT* protocol version used for connection

  typedef 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_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 |

  typedef struct **esp_mqtt_event_t**

  - *MQTT* event configuration structure
typedef esp_mqtt_event_t *esp_mqtt_event_handle_t

typedef struct esp_mqtt_client_config_t esp_mqtt_client_config_t

MQTT client configuration structure

- Default values can be set via menuconfig
- All certificates and key data could be passed in PEM or DER format. PEM format must have a terminating NULL character and the related len field set to 0. DER format requires a related len field set to the correct length.

typedef struct topic_t esp_mqtt_topic_t

Topic definition struct

Enumerations

enum esp_mqtt_event_id_t

MQTT event types.

User event handler receives context data in esp_mqtt_event_t structure with
- client - MQTT client handle
- various other data depending on event type

Values:

enumerator MQTT_EVENT_ANY

enumerator MQTT_EVENT_ERROR

on error event, additional context: connection return code, error handle from esp_tls (if supported)

enumerator MQTT_EVENT_CONNECTED

connected event, additional context: session_present flag

enumerator MQTT_EVENT_DISCONNECTED

disconnected event

enumerator MQTT_EVENT_SUBSCRIBED

subscribed event, additional context:
- msg_id message id
- error_handle error_type in case subscribing failed
- data pointer to broker response, check for errors.
- data_len length of the data for this event

enumerator MQTT_EVENT_UNSUBSCRIBED

unsubscribed event, additional context: msg_id

enumerator MQTT_EVENT_PUBLISHED

published event, additional context: msg_id

enumerator 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
• retain retain flag of the message
• qos QoS level of the message
• dup dup flag of the message 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.

enumerator MQTT_EVENT_BEFORE_CONNECT
The event occurs before connecting

enumerator 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).

enumerator MQTT_USER_EVENT
Custom event used to queue tasks into mqtt event handler All fields from the esp_mqtt_event_t type could be used to pass an additional context data to the handler.

enum esp_mqtt_connect_return_code_t
MQTT connection error codes propagated via ERROR event

Values:

enumerator MQTT_CONNECTION_ACCEPTED
Connection accepted

enumerator MQTT_CONNECTION_REFUSE_PROTOCOL
MQTT connection refused reason: Wrong protocol

enumerator MQTT_CONNECTION_REFUSE_ID_REJECTED
MQTT connection refused reason: ID rejected

enumerator MQTT_CONNECTION_REFUSE_SERVER_UNAVAILABLE
MQTT connection refused reason: Server unavailable

enumerator MQTT_CONNECTION_REFUSE_BAD_USERNAME
MQTT connection refused reason: Wrong user

enumerator 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:
Chapter 2. API Reference

enumerator **MQTT_ERROR_TYPE_NONE**

enumerator **MQTT_ERROR_TYPE_TCP_TRANSPORT**

enumerator **MQTT_ERROR_TYPE_CONNECTION_REFUSED**

enumerator **MQTT_ERROR_TYPE_SUBSCRIBE_FAILED**

enum **esp_mqtt_transport_t**

Values:

enumerator **MQTT_TRANSPORT_UNKNOWN**

enumerator **MQTT_TRANSPORT_OVER_TCP**

MQTT over TCP, using scheme: MQTT

enumerator **MQTT_TRANSPORT_OVER_SSL**

MQTT over SSL, using scheme: MQTT

enumerator **MQTT_TRANSPORT_OVER_WS**

MQTT over Websocket, using scheme:: ws

enumerator **MQTT_TRANSPORT_OVER_WSS**

MQTT over Websocket Secure, using scheme: wss

enum **esp_mqtt_protocol_ver_t**

MQTT protocol version used for connection

Values:

enumerator **MQTT_PROTOCOL_UNDEFINED**

enumerator **MQTT_PROTOCOL_V_3_1**

enumerator **MQTT_PROTOCOL_V_3_1_1**

enumerator **MQTT_PROTOCOL_V_5**

### 2.2.4 ESP-TLS

**Overview**

The ESP-TLS component provides a simplified API interface for accessing the commonly used TLS functions. It supports common scenarios like CA certification validation, SNI, ALPN negotiation, and non-blocking connection among others. All the configurations can be specified in the `esp_tls_cfg_t` data structure. Once done, TLS communication can be conducted using the following APIs:
Chapter 2. API Reference

- `esp_tls_init()`: for initializing the TLS connection handle.
- `esp_tls_conn_new_sync()`: for opening a new blocking TLS connection.
- `esp_tls_conn_new_async()`: for opening a new non-blocking TLS connection.
- `esp_tls_conn_read()`: for reading from the connection.
- `esp_tls_conn_write()`: for writing into the connection.
- `esp_tls_conn_destroy()`: 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 contains the public API headers for the component. Internally, the ESP-TLS component operates using either MbedTLS or WolfSSL, which are SSL/TLS libraries. APIs specific to MbedTLS are present in `esp-tls/private_include/esp_tls_mbedtls.h` and APIs specific to WolfSSL are present in `esp-tls/private_include/esp_tls_wolfssl.h`.

**TLS Server Verification**

ESP-TLS provides multiple options for TLS server verification on the client side. The ESP-TLS client can verify the server by validating the peer’s server certificate or with the help of pre-shared keys. The user should select only one of the following options in the `esp_tls_cfg_t` structure for TLS server verification. If no option is selected, the client will return a fatal error by default during the TLS connection setup.

- `cacert_buf` and `cacert_bytes`: The CA certificate can be provided in a buffer to the `esp_tls_cfg_t` structure. The ESP-TLS uses the CA certificate present in the buffer to verify the server. The following variables in the `esp_tls_cfg_t` structure must be set.
  - `cacert_buf` - pointer to the buffer which contains the CA certification.
  - `cacert_bytes` - the size of the CA certificate in bytes.
- `use_global_ca_store`: The `global_ca_store` can be initialized and set at once. Then it can be used to verify the server for all the ESP-TLS connections which have set `use_global_ca_store = true` in their respective `esp_tls_cfg_t` structure. See the API Reference section below for information regarding different APIs used for initializing and setting up the `global_ca_store`.
- `crt_bundleAttach`: The ESP x509 Certificate Bundle API provides an easy way to include a bundle of custom x509 root certificates for TLS server verification. More details can be found at ESP x509 Certificate Bundle.
- `psk_hint_key`: To use pre-shared keys for server verification, `CONFIG_ESP_TLS_PSK_VERIFICATION` should be enabled in the ESP-TLS menuconfig. Then the pointer to the PSK hint and key should be provided to the `esp_tls_cfg_t` structure. The ESP-TLS will use the PSK for server verification only when no other option regarding server verification is selected.
- `skip_server_verification`: This is an insecure option provided in the ESP-TLS for testing purposes. The option can be set by enabling `CONFIG_ESP_TLS_INSECURE` and `CONFIG_ESP_TLS_SKIP_SERVER_CERT_VERIFY` in the ESP-TLS menuconfig. When this option
is enabled the ESP-TLS will skip server verification by default when no other options for server verification are selected in the `esp_tls_cfg_t` structure.

**Warning:** Enabling this option comes with a potential risk of establishing a TLS connection with a server that has a fake identity, provided that the server certificate is not provided either through API or other mechanisms like `ca_store` etc.

### ESP-TLS Server Cert Selection Hook

The ESP-TLS component provides an option to set the server certification selection hook when using the MbedTLS stack. This provides an ability to configure and use a certificate selection callback during server handshake. The callback helps to select a certificate to present to the client based on the TLS extensions supplied in the client hello message, such as ALPN and SNI. To enable this feature, please enable `CONFIG_ESP_TLS_SERVER_CERT_SELECT_HOOK` in the ESP-TLS menuconfig.

The certificate selection callback can be configured in the `esp_tls_cfg_t` structure as follows:

```c
int cert_selection_callback(mbedtls_ssl_context *ssl)
{
    /* Code that the callback should execute */
    return 0;
}

esp_tls_cfg_t cfg = {
    cert_select_cb = cert_section_callback,
};
```

### Underlying SSL/TLS Library Options

The ESP-TLS component offers the option to use MbedTLS or WolfSSL as its underlying SSL/TLS library. By default, only MbedTLS is available and used, WolfSSL SSL/TLS library is also available publicly at https://github.com/espressif/esp-wolfssl. The repository provides the WolfSSL component in binary format, and it also provides a few examples that are useful for understanding the API. Please refer to the repository README.md for information on licensing and other options. Please see the below section for instructions on how to use 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 the following three commands:

```
(First, change the directory (cd) to your project directory)
mkdir components
cd components
git clone --recursive https://github.com/espressif/esp-wolfssl.git
```

2) Add WolfSSL as an extra component in your project.

- Download WolfSSL with:
  ```
git clone --recursive https://github.com/espressif/esp-wolfssl.git
  ```
- Include ESP-WolfSSL in ESP-IDF with setting `EXTRA_COMPONENT_DIRS` in `CMakeLists.txt` of your project as done in `wolfssl/examples`. For reference see `Optional Project Variables` in `build-system`. 
After the above steps, you will have the option to choose WolfSSL as the underlying SSL/TLS library in the configuration menu of your project as follows:

```
idf.py menuconfig > ESP-TLS > SSL/TLS Library > Mbedtls/Wolfssl
```

### Comparison Between MbedTLS and WolfSSL

The following table shows a typical comparison between WolfSSL and MbedTLS when the `protocols/https_request` example (which includes server authentication) is running with both SSL/TLS libraries and with all respective configurations set to default. For MbedTLS, the `IN_CONTENT` length and `OUT_CONTENT` length are set to 16384 bytes and 4096 bytes respectively.

<table>
<thead>
<tr>
<th>Property</th>
<th>WolfSSL</th>
<th>MbedTLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Heap Consumed</td>
<td>~19 KB</td>
<td>~37 KB</td>
</tr>
<tr>
<td>Task Stack Used</td>
<td>~2.2 KB</td>
<td>~3.6 KB</td>
</tr>
<tr>
<td>Bin size</td>
<td>~858 KB</td>
<td>~736 KB</td>
</tr>
</tbody>
</table>

**Note:** These values can vary based on configuration options and version of respective libraries.

### Digital Signature with ESP-TLS

ESP-TLS provides support for using the Digital Signature (DS) with ESP32-S2. Use of the DS for TLS is supported only when ESP-TLS is used with MbedTLS (default stack) as its underlying SSL/TLS stack. For more details on Digital Signature, please refer to the [Digital Signature (DS)](https://esp32-s2-technical-reference-manual.pdf). The technical details of Digital Signature such as how to calculate private key parameters can be found in [ESP32-S2 Technical Reference Manual > Digital Signature (DS)](https://esp32-s2-technical-reference-manual.pdf). The DS peripheral must be configured before it can be used to perform Digital Signature, see Configure the DS Peripheral for a TLS Connection.

The DS peripheral must be initialized with the required encrypted private key parameters, which are obtained when the DS peripheral is configured. ESP-TLS internally initializes the DS peripheral when provided with the required DS context, i.e., DS parameters. Please see the below code snippet for passing the DS context to the ESP-TLS context. The DS context passed to the ESP-TLS context should not be freed till the TLS connection is deleted.

```c
#include "esp_tls.h"

esp_ds_data_ctx_t *ds_ctx;
/* initialize ds_ctx with encrypted private key parameters, which can be read from...*/
esp_tls_cfg_t cfg = {
    .clientcert_buf = /* the client certification */,
    .clientcert_bytes = /* length of the client certification */,
    /* other configurations options */
    .ds_data = (void *)ds_ctx,
};
```

**Note:** When using Digital Signature for the TLS connection, along with the other required params, only the client certification (`clientcert_buf`) and the DS params (`ds_data`) are required and the client key (`clientkey_buf`) can be set to NULL.

- An example of mutual authentication with the DS peripheral can be found at `ssl mutual auth` which internally uses (ESP-TLS) for the TLS connection.
TLS Ciphersuites

ESP-TLS provides the ability to set a ciphersuites list in client mode. The TLS ciphersuites list informs the server about the supported ciphersuites for the specific TLS connection regardless of the TLS stack configuration. If the server supports any ciphersuite from this list, then the TLS connection will succeed; otherwise, it will fail.

You can set ciphersuites_list in the `esp_tls_cfg_t` structure during client connection as follows:

```c
/* ciphersuites_list must end with 0 and must be available in the memory scope active during the entire TLS connection */
static const int ciphersuites_list[] = {
    MBEDTLS_TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
    MBEDTLS_TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
    0};
```

ESP-TLS will not check the validity of `ciphersuites_list` that was set, you should call `esp_tls_get_ciphersuites_list()` to get ciphersuites list supported in the TLS stack and cross-check it against the supplied list.

**Note:** This feature is supported only in the MbedTLS stack.

TLS Protocol Version

ESP-TLS provides the ability to set the TLS protocol version for the respective TLS connection. Once the version is specified, it should be exclusively used to establish the TLS connection. This provides an ability to route different TLS connections to different protocol versions like TLS 1.2 and TLS 1.3 at runtime.

**Note:** At the moment, the feature is supported only when ESP-TLS is used with MbedTLS as its underlying SSL/TLS stack.

To set TLS protocol version with ESP-TLS, set `esp_tls_cfg_t::tls_version` to the required protocol version from `esp_tls_proto_ver_t`. If the protocol version field is not set, then the default policy is to allow TLS connection based on the server requirement.

The ESP-TLS connection can be configured to use the specified protocol version as follows:

```c
#include "esp_tls.h"
esp_tls_cfg_t cfg = {
    .tls_version = ESP_TLS_VER_TLS_1_2,
};
```

API Reference

**Header File**

- `components/esp-tls/esp_tls.h`
- This header file can be included with:

  ```c
  #include "esp_tls.h"
  ```
- This header file is a part of the API provided by the `esp-tls` component. To declare that your component depends on `esp-tls`, add the following to your CMakeLists.txt:

  ```cmake
  REQUIREES esp-tls
  ```
Chapter 2. API Reference

PRIV_REQUIRES esp-tls

Functions

```c
esp_err_t esp_tls_cfg_server_session_tickets_init (esp_tls_cfg_server_t *cfg)
```

Initialize the server side TLS session ticket context.

This function initializes the server side TLS session ticket context which holds all necessary data structures to enable TLS session tickets according to RFC5077. Use `esp_tls_cfg_server_session_tickets_free` to free the data.

- **Parameters**
  - `cfg` [in] server configuration as `esp_tls_cfg_server_t`
- **Returns**
  - `ESP_OK` if setup succeeded
  - `ESP_ERR_INVALID_ARG` if context is already initialized
  - `ESP_ERR_NO_MEM` if memory allocation failed
  - `ESP_ERR_NOT_SUPPORTED` if session tickets are not available due to build configuration
  - `ESP_FAIL` if setup failed

```c
void esp_tls_cfg_server_session_tickets_free (esp_tls_cfg_server_t *cfg)
```

Free the server side TLS session ticket context.

- **Parameters**
  - `cfg` [in] server configuration as `esp_tls_cfg_server_t`

```c
esp_tls_t *esp_tls_init (void)
```

Create TLS connection.

This function allocates and initializes esp-tls structure handle.

- **Returns**
  - `tls` Pointer to esp-tls as esp-tls handle if successfully initialized, `NULL` if allocation error

```c
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.

- **Parameters**
  - `url` [in] url of host.
  - `cfg` [in] 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.
- **Returns**
  - Pointer to `esp_tls_t`, or `NULL` if connection couldn’t be opened.

```c
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.

- **Parameters**
  - `hostname` [in] Hostname of the host.
  - `hostlen` [in] Length of hostname.
  - `port` [in] Port number of the host.
  - `cfg` [in] 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.
  - `tls` [in] Pointer to esp-tls as esp-tls handle.
- **Returns**
  - `-1` If connection establishment fails.
  - `1` If connection establishment is successful.
  - `0` If connection state is in progress.

```c
int esp_tls_conn_http_new_sync (const char *url, const esp_tls_cfg_t *cfg, esp_tls_t *tls)
```

Create a new blocking TLS/SSL connection with a given “HTTP” url.

The behaviour is same as `esp_tls_conn_new_sync()` API. However this API accepts host’s url.
Chapter 2. API Reference

Parameters
- **url** – [in] url of host.
- **cfg** – [in] 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.
- **tls** – [in] Pointer to esp-tls as esp-tls handle.

Returns
- -1 If connection establishment fails.
- 1 If connection establishment is successful.
- 0 If connection state is in progress.

```c
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.

Parameters
- **hostname** – [in] Hostname of the host.
- **hostlen** – [in] Length of hostname.
- **port** – [in] Port number of the host.
- **cfg** – [in] TLS configuration as esp_tls_cfg_t. `non_block` member of this structure should be set to be true.
- **tls** – [in] pointer to esp-tls as esp-tls handle.

Returns
- -1 If connection establishment fails.
- 0 If connection establishment is in progress.
- 1 If connection establishment is successful.

```c
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_async() API. However this API accepts host’s url.

Parameters
- **url** – [in] url of host.
- **cfg** – [in] TLS configuration as esp_tls_cfg_t.
- **tls** – [in] pointer to esp-tls as esp-tls handle.

Returns
- -1 If connection establishment fails.
- 0 If connection establishment is in progress.
- 1 If connection establishment is successful.

```c
ssize_t esp_tls_conn_write (esp_tls_t *tls, const void *data, size_t datalen)
```

Write from buffer ‘data’ into specified tls connection.

Parameters
- **tls** – [in] pointer to esp-tls as esp-tls handle.
- **data** – [in] Buffer from which data will be written.
- **datalen** – [in] Length of data buffer.

Returns
- >=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, 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.

```c
ssize_t esp_tls_conn_read (esp_tls_t *tls, void *data, size_t datalen)
```

Read from specified tls connection into the buffer ‘data’.
Chapter 2. API Reference

Parameters
- `tls` [in] pointer to esp-tls as esp-tls handle.
- `data` [in] Buffer to hold read data.
- `datalen` [in] Length of data buffer.

Returns
- >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.

```c
int esp_tls_conn_destroy(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_sync()` (or `esp_tls_conn_http_new_sync()`) and `esp_tls_conn_new_async()` (or `esp_tls_conn_http_new_async()`) APIs.

**Parameters**
- `tls` [in] pointer to esp-tls as esp-tls handle.

**Returns**
- 0 on success
- -1 if socket error or an invalid argument

```c
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.

**Parameters**
- `tls` [in] pointer to esp-tls as esp-tls handle.

**Returns**
- -1 in case of invalid arg
- bytes available in the application data record read buffer

```c
esp_err_t esp_tls_get_conn_sockfd(esp_tls_t *tls, int *sockfd)
```

Returns the connection socket file descriptor from esp_tls session.

**Parameters**
- `tls` [in] handle to esp_tls context
- `sockfd` [out] int pointer to sockfd value.

**Returns**
- ESP_OK on success and value of sockfd will be updated with socket file descriptor for connection
- ESP_ERR_INVALID_ARG if (tls == NULL || sockfd == NULL)

```c
esp_err_t esp_tls_set_conn_sockfd(esp_tls_t *tls, int sockfd)
```

Sets the connection socket file descriptor for the esp_tls session.

**Parameters**
- `tls` [in] handle to esp_tls context
- `sockfd` [in] sockfd value to set.

**Returns**
- ESP_OK on success and value of sockfd for the tls connection shall updated with the provided value
- ESP_ERR_INVALID_ARG if (tls == NULL || sockfd < 0)

```c
esp_err_t esp_tls_get_conn_state(esp_tls_t *tls, esp_tls_conn_state_t *conn_state)
```

Gets the connection state for the esp_tls session.

**Parameters**
- `tls` [in] handle to esp_tls context
- `conn_state` [out] pointer to the connection state value.

**Returns**
- ESP_OK on success and value of sockfd for the tls connection shall updated with the provided value
- ESP_ERR_INVALID_ARG (Invalid arguments)

```c
esp_err_t esp_tls_set_conn_state(esp_tls_t *tls, esp_tls_conn_state_t conn_state)
```

Sets the connection state for the esp_tls session.
## Parameters

- **tls** - [in] handle to esp tls context
- **conn_state** - [in] connection state value to set.

## Returns

- ESP_OK on success and value of sockfd for the tls connection shall updated with the provided value
- ESP_ERR_INVALID_ARG (Invalid arguments)

### void *esp_tls_get_ssl_context (esp_tls_t *tls)

Returns the ssl context.

### Parameters

- **tls** - [in] handle to esp_tls context

### Returns

- ssl_ctx pointer to ssl context of underlying TLS layer on success
- NULL in case of error

### 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().

### Returns

- 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().

### Parameters

- **cacert_pem_buf** - [in] 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.
- **cacert_pem_bytes** - [in] Length of the buffer.

### Returns

- ESP_OK if adding certificates was successful.
- Other if an error occurred or an action must be taken by the calling process.

### 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.

### Parameters

- **h** - [in] esp-tls error handle.
- **esp_tls_code** - [out] 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
- **esp_tls_flags** - [out] last certification verification flags (set to zero if none) This pointer could be NULL if caller does not care about esp_tls_code

### Returns

- 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
**esp_err_t esp_tls_get_and_clear_error_type** *(esp_tls_error_handle_t h, esp_tls_error_type_t err_type, int *error_code)*

Returns the last error captured in esp_tls of a specific type. The error information is cleared internally upon return.

**Parameters**
- **h** – [in] esp-tls error handle.
- **err_type** – [in] specific error type
- **error_code** – [out] 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

**Returns**
- ESP_ERR_INVALID_STATE if invalid parameters
- ESP_OK if a valid error returned and was cleared

**esp_err_t esp_tls_get_error_handle** *(esp_tls_t *tls, esp_tls_error_handle_t *error_handle)*

Returns the ESP-TLS error_handle.

**Parameters**
- **tls** – [in] handle to esp_tls context
- **error_handle** – [out] pointer to the error handle.

**Returns**
- ESP_OK on success and error_handle will be updated with the ESP-TLS error handle.
- ESP_ERR_INVALID_ARG if (tls==NULL || error_handle==NULL)

**mbedtls_x509_crt** *esp_tls_get_global_ca_store (void)*

Get the pointer to the global CA store currently being used.

The application must first call esp_tls_set_global_ca_store(). Then the same CA store could be used by the application for APIs other than esp_tls.

**Note:** Modifying the pointer might cause a failure in verifying the certificates.

**const int** *esp_tls_get_ciphersuites_list (void)*

Get supported TLS ciphersuites list.

See [https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4](https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4) for the list of ciphersuites

**Returns** Pointer to a zero-terminated array of IANA identifiers of TLS ciphersuites.

**int esp_tls_server_session_create** *(esp_tls_cfg_server_t *cfg, int sockfd, esp_tls_t *tls)*

Create TLS/SSL server session.

This function creates a TLS/SSL server context for already accepted client connection and performs TLS/SSL handshake with the client

**Parameters**
- **cfg** – [in] Pointer to esp_tls_cfg_server_t
- **sockfd** – [in] FD of accepted connection
- **tls** – [out] Pointer to allocated esp_tls_t

**Returns**
- 0 if successful
- <0 in case of error

**void esp_tls_server_session_delete** *(esp_tls_t *tls)*

Close the server side TLS/SSL connection and free any allocated resources.
This function should be called to close each tls connection opened with esp_tls_server_session_create()

```c
esp_err_t esp_tls_plain_tcp_connect (const char* host, int hostlen, int port, const esp_tls_cfg_t *cfg,
        esp_tls_error_handle_t error_handle, int *sockfd)
```

Creates a plain TCP connection, returning a valid socket fd on success or an error handle.

**Parameters**
- `host` - [in] Hostname of the host.
- `hostlen` - [in] Length of hostname.
- `port` - [in] Port number of the host.
- `cfg` - [in] ESP-TLS configuration as esp_tls_cfg_t.
- `error_handle` - [out] ESP-TLS error handle holding potential errors occurred during connection.
- `sockfd` - [out] Socket descriptor if successfully connected on TCP layer.

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if invalid output parameters
- ESP-TLS based error codes on failure

**Structures**

```c
struct psk_key_hint
```

ESP-TLS pre-shared 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.

```c
struct tls_keep_alive_cfg
```

esp-tls client session ticket ctx

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 mbedtlss-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 use_ecdsa_peripheral
    Use the ECDSA peripheral for the private key operations

uint8_t ecdsa_key_efuse_blk
    The efuse block where the ECDSA key is stored

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

bool use_secure_element
    Enable this option to use secure element or atecc608a chip (Integrated with ESP32-WROOM-32SE)

int timeout_ms
    Network timeout in milliseconds. Note: If this value is not set, by default the timeout is set to 10 seconds. If you wish that the session should wait indefinitely then please use a larger value e.g., INT32_MAX

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

 tls_keep_alive_cfg_t *keep_alive_cfg
    Enable TCP keep-alive timeout for SSL connection
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.

esp_err_t (*crt_bundle_attach)(void *conf)

Function pointer to esp_crt_bundle_attach. Enables the use of certification bundle for server verification, must be enabled in menuconfig.

void *ds_data

Pointer for digital signature peripheral context.

bool is_plain_tcp

Use non-TLS connection: When set to true, the esp-tls uses plain TCP transport rather than TLS/SSL connection. Note, that it is possible to connect using a plain tcp transport directly with esp_tls_plain_tcp_connect() API.

struct ifreq *if_name

The name of interface for data to go through. Use the default interface without setting.

dl_data

Pointer for digital signature peripheral context.

const esp_tls_cfg_server addr_family

The address family to use when connecting to a host.

const int *ciphersuites_list

Pointer to a zero-terminated array of IANA identifiers of TLS ciphersuites. Please check the list validity by esp_tls_get_ciphersuites_list() API.

dl_data

TLS protocol version of the connection, e.g., TLS 1.2, TLS 1.3 (default - no preference).

struct esp_tls_cfg_server

ESP-TLS Server configuration parameters.

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

Client CA certificate in a buffer. This buffer should be NULL terminated.

const unsigned char *cacert_pem_buf

Client CA certificate legacy name.

unsigned int cacert_bytes

Size of client CA certificate pointed to by cacert_pem_buf.
unsigned int `cacert_pem_bytes`
  Size of client CA certificate legacy name

const unsigned char *`servercert_buf`
  Server certificate in a buffer This buffer should be NULL terminated

const unsigned char *`servercert_pem_buf`
  Server certificate legacy name

unsigned int `servercert_bytes`
  Size of server certificate pointed to by servercert_pem_buf

unsigned int `servercert_pem_bytes`
  Size of server certificate legacy name

const unsigned char *`serverkey_buf`
  Server key in a buffer This buffer should be NULL terminated

const unsigned char *`serverkey_pem_buf`
  Server key legacy name

unsigned int `serverkey_bytes`
  Size of server key pointed to by serverkey_pem_buf

unsigned int `serverkey_pem_bytes`
  Size of server key legacy name

const unsigned char *`serverkey_password`
  Server key decryption password string

unsigned int `serverkey_password_len`
  String length of the password pointed to by serverkey_password

bool `use_ecdsa_peripheral`
  Use ECDSA peripheral to use private key

uint8_t `ecdsa_key_efuse_blk`
  The efuse block where ECDSA key is stored

bool `use_secure_element`
  Enable this option to use secure element or atec608a chip (Integrated with ESP32-WROOM-32SE)

void *`userdata`
  User data to be added to the ssl context. Can be retrieved by callbacks

**Type Definitions**

typedef enum `esp_tls_conn_state` esptlsconnstate_t
  ESP-TLS Connection State.
typedef enum \texttt{esp\_tls\_role} \texttt{esp\_tls\_role\_t}

typedef struct \texttt{psk\_key\_hint} \texttt{psk\_hint\_key\_t}

ESP-TLS preshared key and hint structure.

typedef struct \texttt{tls\_keep\_alive\_cfg} \texttt{tls\_keep\_alive\_cfg\_t}

esp-tls client session ticket ctx

Keep alive parameters structure

typedef enum \texttt{esp\_tls\_addr\_family} \texttt{esp\_tls\_addr\_family\_t}

typedef struct \texttt{esp\_tls\_cfg} \texttt{esp\_tls\_cfg\_t}

ESP-TLS configuration parameters.

\textbf{Note:} Note about format of certificates:

\begin{itemize}
\item 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).
\item 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.
\item 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.
\end{itemize}

typedef struct \texttt{esp\_tls\_cfg\_server} \texttt{esp\_tls\_cfg\_server\_t}

ESP-TLS Server configuration parameters.

typedef struct \texttt{esp\_tls} \texttt{esp\_tls\_t}

\textbf{Enumerations}

\begin{verbatim}
enum \texttt{esp\_tls\_conn\_state} \texttt{esp\_tls\_conn\_state\_t}

ESP-TLS Connection State.

Values:

enumerator \texttt{ESP\_TLS\_INIT}

enumerator \texttt{ESP\_TLS\_CONNECTING}

enumerator \texttt{ESP\_TLS\_HANDSHAKE}

enumerator \texttt{ESP\_TLS\_FAIL}

enumerator \texttt{ESP\_TLS\_DONE}

enum \texttt{esp\_tls\_role} \texttt{esp\_tls\_role\_t}

Values:
\end{verbatim}
enumerator ESP_TLS_CLIENT

enumerator ESP_TLS_SERVER

enum esp_tls_addr_family
    
Values:

enumerator ESP_TLS_AF_UNSPEC
    Unspecified address family.

enumerator ESP_TLS_AF_INET
    IPv4 address family.

enumerator ESP_TLS_AF_INET6
    IPv6 address family.

enum esp_tls_proto_ver_t
    
Values:

enumerator ESP_TLS_VER_ANY

enumerator ESP_TLS_VER_TLS_1_2

enumerator ESP_TLS_VER_TLS_1_3

enumerator ESP_TLS_VER_TLS_MAX

Header File

• components/esp-tls/esp_tls_errors.h
• This header file can be included with:

```c
#include "esp_tls_errors.h"
```

• This header file is a part of the API provided by the esp-tls component. To declare that your component depends on esp-tls, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp-tls
```

or

```cmake
PRIV_REQUIRES esp-tls
```

Structures

struct esp_tls_last_error

Error structure containing relevant errors in case tls error occurred.

Public Members
Chapter 2. API Reference

```
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

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/get socket option

ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT
   new connection in esp_tls_low_level_conn connection timeouted

ESP_ERR_ESP_TLS_SE_FAILED

ESP_ERR_ESP_TLS_TCP_CLOSED_FIN

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_MBEDTLS_SSL_TICKET_SETUP_FAILED
    mbedtls api returned failed

ESP_ERR_WOLFSSL_SSL_SET_HOSTNAME_FAILED
    wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED
    wolfSSL api returned error

ESP_ERR_WOLFSSL_CERT_VERIFY_SETUP_FAILED
    wolfSSL api returned error

ESP_ERR_WOLFSSL_KEY_VERIFY_SETUP_FAILED
    wolfSSL api returned error

ESP_ERR_WOLFSSL_SSL_HANDSHAKE_FAILED
    wolfSSL api returned failed

ESP_ERR_WOLFSSL_CTX_SETUP_FAILED
    wolfSSL api returned failed

ESP_ERR_WOLFSSL_SSL_SETUP_FAILED
    wolfSSL api returned failed
Chapter 2.  API Reference

ESP_ERR_WOLFSSL_SSL_WRITE_FAILED
wolfSSL api returned failed

ESP_TLS_ERR_SSL_WANT_READ
Definition of errors reported from IO API (potentially non-blocking) in case of error:
• esp_tls_conn_read()
  • esp_tls_conn_write()

ESP_TLS_ERR_SSL_WANT_WRITE

ESP_TLS_ERR_SSL_TIMEOUT

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.

Enumerations
enum esp_tls_error_type_t
Definition of different types/sources of error codes reported from different components
Values:

enumerator ESP_TLS_ERR_TYPE_UNKNOWN
enumerator ESP_TLS_ERR_TYPE_SYSTEM
  System error &#8212; errno
enumerator ESP_TLS_ERR_TYPE_MBEDTLS
  Error code from mbedTLS library
enumerator ESP_TLS_ERR_TYPE_MBEDTLS_CERT_FLAGS
  Certificate flags defined in mbedTLS
enumerator ESP_TLS_ERR_TYPE_ESP
  ESP-IDF error type &#8212; esp_err_t
enumerator ESP_TLS_ERR_TYPE_WOLFSSL
  Error code from wolfSSL library
enumerator ESP_TLS_ERR_TYPE_WOLFSSL_CERT_FLAGS
  Certificate flags defined in wolfSSL
enumerator ESP_TLS_ERR_TYPE_MAX
  Last err type &#8212; invalid entry
2.2.5 ESP HTTP Client

Overview

`esp_http_client` component provides a set of APIs for making HTTP/S requests from ESP-IDF applications. The steps to use these APIs are as follows:

- `esp_http_client_init()`: Creates an `esp_http_client_handle_t` instance, i.e., an HTTP client handle based on the given `esp_http_client_config_t` configuration. This function must be the first to be called; default values are assumed for the configuration values that are not explicitly defined by the user.
- `esp_http_client_perform()`: Performs all operations of the `esp_http_client` - opening the connection, exchanging data, and closing the connection (as required), while blocking the current task before its completion. All related events are invoked through the event handler (as specified in `esp_http_client_config_t`).
- `esp_http_client_cleanup()`: Closes the connection (if any) and frees up all the memory allocated to the HTTP client instance. This must be the last function to be called after the completion of operations.

Application Example

Simple example that uses ESP HTTP Client to make HTTP/S requests can be found at `protocols/esp_http_client`.

Basic HTTP Request

Check out the example functions `http_rest_with_url` and `http_rest_with_hostname_path` in the application example for implementation details.

Persistent Connections

Persistent connection means that the HTTP client can re-use the same connection for several exchanges. If the server does not request to close the connection with the `Connection: close` header, the connection is not dropped but is instead kept open and used for further requests.

To allow ESP HTTP client to take full advantage of persistent connections, one should make as many requests as possible using the same handle instance. Check out the example functions `http_rest_with_url` and `http_rest_with_hostname_path` in the application example. Here, once the connection is created, multiple requests (GET, POST, PUT, etc.) are made before the connection is closed.

HTTPS Request

ESP HTTP client supports SSL connections using `mbedtls`, with the `url` configuration starting with `https` scheme or `transport_type` set to `HTTP_TRANSPORT_OVER_SSL`. HTTPS support can be configured via `CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS` (enabled by default).

**Note:** While making HTTPS requests, if server verification is needed, an additional root certificate (in PEM format) needs to be provided to the `cert_pem` member in the `esp_http_client_config_t` configuration. Users can also use the ESP x509 Certificate Bundle for server verification using the `crt_bundle_attach` member of the `esp_http_client_config_t` configuration.

Check out the example functions `https_with_url` and `https_with_hostname_path` in the application example for implementation details of the above note.
HTTP Stream

Some applications need to open the connection and control the exchange of data actively (data streaming). In such cases, the application flow is different from regular requests. Example flow is given below:

- `esp_http_client_init()`: Create a HTTP client handle.
- `esp_http_client_set_*` or `esp_http_client_delete_*`: Modify the HTTP connection parameters (optional).
- `esp_http_client_open()`: Open the HTTP connection with `write_len` parameter (content length that needs to be written to server), set `write_len=0` for read-only connection.
- `esp_http_client_write()`: Write data to server with a maximum length equal to `write_len` of `esp_http_client_open()` function; no need to call this function for `write_len=0`.
- `esp_http_client_fetch_headers()`: Read the HTTP Server response headers, after sending the request headers and server data (if any). Returns the `content-length` from the server and can be succeeded by `esp_http_client_get_status_code()` for getting the HTTP status of the connection.
- `esp_http_client_read()`: Read the HTTP stream.
- `esp_http_client_close()`: Close the connection.
- `esp_http_client_cleanup()`: Release allocated resources.

Check out the example function `http_perform_as_stream_reader` in the application example for implementation details.

HTTP Authentication

ESP HTTP client supports both Basic and Digest Authentication.

- Users can provide the username and password in the `url` or the `username` and `password` members of the `esp_http_client_config_t` configuration. For `auth_type = HTTP_AUTH_TYPE_BASIC`, the HTTP client takes only one perform operation to pass the authentication process.
- If `auth_type = HTTP_AUTH_TYPE_NONE`, but the `username` and `password` fields are present in the configuration, the HTTP client takes two perform operations. The client will receive the 401 Unauthorized header in its first attempt to connect to the server. Based on this information, it decides which authentication method to choose and performs it in the second operation.
- Check out the example functions `http_auth_basic`, `http_auth_basic_redirect` (for Basic authentication) and `http_auth_digest` (for Digest authentication) in the application example for implementation details.

Examples of Authentication Configuration

- Authentication with URI
  ```c
  esp_http_client_config_t config = {
      .url = "http://user:passwd@httpbin.org/basic-auth/user/passwd",
      .auth_type = HTTP_AUTH_TYPE_BASIC,
  };
  ```

- Authentication with username and password entry
  ```c
  esp_http_client_config_t config = {
      .url = "http://httpbin.org/basic-auth/user/passwd",
      .username = "user",
      .password = "passwd",
      .auth_type = HTTP_AUTH_TYPE_BASIC,
  };
  ```

Event Handling

ESP HTTP Client supports event handling by triggering an event handler corresponding to the event which takes place. `esp_http_client_event_id_t` contains all the events which could occur while performing an HTTP
request using the ESP HTTP Client.

To enable event handling, you just need to set a callback function using the `esp_http_client_config_t::event_handler` member.

**ESP HTTP Client Diagnostic Information**

Diagnostic information could be helpful to gain insights into a problem. In the case of ESP HTTP Client, the diagnostic information can be collected by registering an event handler with the *Event Loop library*. This feature has been added by keeping in mind the ESP Insights framework which collects the diagnostic information. However, this feature can also be used without any dependency on the ESP Insights framework for the diagnostic purpose. Event handler can be registered to the event loop using the `esp_event_handler_register()` function.

Expected data types for different HTTP Client events in the event loop are as follows:

- **HTTP_EVENT_ERROR**: `esp_http_client_handle_t`
- **HTTP_EVENT_ON_CONNECTED**: `esp_http_client_handle_t`
- **HTTP_EVENT_HEADERS_SENT**: `esp_http_client_handle_t`
- **HTTP_EVENT_ON_HEADER**: `esp_http_client_handle_t`
- **HTTP_EVENT_ON_DATA**: `esp_http_client_on_data_t`
- **HTTP_EVENT_ON_FINISH**: `esp_http_client_handle_t`
- **HTTP_EVENT_DISCONNECTED**: `esp_http_client_handle_t`
- **HTTP_EVENT_REDIRECT**: `esp_http_client_redirect_event_data_t`

The `esp_http_client_handle_t` received along with the event data will be valid until `HTTP_EVENT_DISCONNECTED` is not received. This handle has been sent primarily to differentiate between different client connections and must not be used for any other purpose, as it may change based on client connection state.

**TLS Protocol Version**

TLS protocol version to be used for the underlying TLS connection can be set in `esp_http_client_config_t`. Please refer to the *TLS Protocol Version* section in the ESP-TLS for more details.

The TLS protocol version for the HTTP client can be configured as follows:

```c
#include "esp_http_client.h"
esp_http_client_config_t config = {
    .tls_version = ESP_HTTP_CLIENT_TLS_VER_TLS_1_2,
};
```

**API Reference**

**Header File**

- components/esp_http_client/include/esp_http_client.h
- This header file can be included with:

  ```c
  #include "esp_http_client.h"
  ```

  This header file is a part of the API provided by the `esp_http_client` component. To declare that your component depends on `esp_http_client`, add the following to your CMakeLists.txt:

  ```cmake
  REQUIRES esp_http_client
  ```

  or

  ```cmake
  PRIV_REQUIRES esp_http_client
  ```
Chapter 2. API Reference

**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.

**Parameters**

- `config` - [in] The configurations, see `http_client_config_t`

**Returns**

- `esp_http_client_handle_t`
- NULL if any errors

**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 invoke 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`.

**Parameters**

- `client` - The `esp_http_client_handle`

**Returns**

- ESP_OK on successful
- ESP_FAIL on error

**esp_err_t esp_http_client_cancel_request(esp_http_client_handle_t client)**

Cancel an ongoing HTTP request. This API closes the current socket and opens a new socket with the same `esp_http_client` context.

**Parameters**

- `client` - The `esp_http_client_handle`

**Returns**

- ESP_OK on successful
- ESP_FAIL on error
- ESP_ERR_INVALID_ARG
- ESP_ERR_INVALID_STATE

**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.

**Parameters**

- `client` - [in] The `esp_http_client_handle`
- `url` - [in] The url

**Returns**

- ESP_OK
- ESP_FAIL
**esp_err_t esp_http_client_set_post_field** *(esp_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.

**Parameters**
- `client` *[in]* The esp_http_client handle
- `data` *[in]* post data pointer
- `len` *[in]* post length

**Returns**
- ESP_OK
- ESP_FAIL

**int esp_http_client_get_post_field** *(esp_client_handle_t client, char**data)*

Get current post field information.

**Parameters**
- `client` *[in]* The client
- `data` *[out]* Point to post data pointer

**Returns** Size of post data

**esp_err_t esp_http_client_set_header** *(esp_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.

**Parameters**
- `client` *[in]* The esp_http_client handle
- `key` *[in]* The header key
- `value` *[in]* The header value

**Returns**
- ESP_OK
- ESP_FAIL

**esp_err_t esp_http_client_get_header** *(esp_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*.

**Parameters**
- `client` *[in]* The esp_http_client handle
- `key` *[in]* The header key
- `value` *[out]* The header value

**Returns**
- ESP_OK
- ESP_FAIL

**esp_err_t esp_http_client_get_username** *(esp_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*.

**Parameters**
- `client` *[in]* The esp_http_client handle
- `value` *[out]* The username value

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**esp_err_t esp_http_client_set_username** *(esp_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.

**Parameters**
- `client` *[in]* The esp_http_client handle
• **username**  - [in] The username value

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**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.

**Parameters**
- **client**  - [in] The esp_http_client handle
- **value**  - [out] The password value

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**esp_err_t esp_http_client_set_password** (esp_http_client_handle_t client, const 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.

**Parameters**
- **client**  - [in] The esp_http_client handle
- **password**  - [in] The password value

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**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.

**Parameters**
- **client**  - [in] The esp_http_client handle
- **auth_type**  - [in] The esp_http_client auth type

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**esp_err_t esp_http_client_get_user_data** (esp_http_client_handle_t client, void **data)

Get http request user_data. The value stored from the esp_http_client_config_t will be written to the address passed into data.

**Parameters**
- **client**  - [in] The esp_http_client handle
- **data**  - [out] A pointer to the pointer that will be set to user_data.

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**esp_err_t esp_http_client_set_user_data** (esp_http_client_handle_t client, void *data)

Set http request user_data. The value passed in *data* will be available during event callbacks. No memory management will be performed on the user’s behalf.

**Parameters**
- **client**  - [in] The esp_http_client handle
- **data**  - [in] The pointer to the user data

**Returns**
- ESP_OK
- ESP_ERR_INVALID_ARG

**int esp_http_client_get_errno** (esp_http_client_handle_t client)

Get HTTP client session errno.
Parameters **client** - [in] The esp_http_client handle

Returns
- (-1) if invalid argument
- errno

```c
esp_err_t esp_http_client_set_method(esp_http_client_handle_t client, esp_http_client_method_t method)
```

Set http request method.

Parameters
- **client** - [in] The esp_http_client handle
- **method** - [in] The method

Returns
- ESP_OK
- ESP_ERR_INVALID_ARG

```c
esp_err_t esp_http_client_set_timeout_ms(esp_http_client_handle_t client, int timeout_ms)
```

Set http request timeout.

Parameters
- **client** - [in] The esp_http_client handle
- **timeout_ms** - [in] The timeout value

Returns
- ESP_OK
- ESP_ERR_INVALID_ARG

```c
esp_err_t esp_http_client_delete_header(esp_http_client_handle_t client, const char* key)
```

Delete http request header.

Parameters
- **client** - [in] The esp_http_client handle
- **key** - [in] The key

Returns
- ESP_OK
- ESP_FAIL

```c
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.

Parameters
- **client** - [in] The esp_http_client handle
- **write_len** - [in] HTTP Content length need to write to the server

Returns
- ESP_OK
- ESP_FAIL

```c
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().

Parameters
- **client** - [in] The esp_http_client handle
- **buffer** - The buffer
- **len** - [in] This value must not be larger than the write_len parameter provided to esp_http_client_open()

Returns
- (-1) if any errors
- Length of data written

```c
int64_t 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.

Parameters **client** - [in] The esp_http_client handle

Returns
Chapter 2. API Reference

- (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
- (-ESP_ERR_HTTP_EAGAIN = -0x7007) if call is timed-out before any data was ready
- Download data length defined by content-length header

bool esp_http_client_is_chunked_response(esp_http_client_handle_t client)
Check response data is chunked.

Parameters
- client -[in] The esp_http_client handle

Returns
- true or false

int esp_http_client_read(esp_http_client_handle_t client, char*buffer, int len)
Read data from http stream.

Note: (-ESP_ERR_HTTP_EAGAIN = -0x7007) is returned when call is timed-out before any data was ready

Parameters
- client -[in] The esp_http_client handle
- buffer -The buffer
- len -[in] The length

Returns
- (-1) if any errors
- Length of data was read

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

Parameters
- client -[in] The esp_http_client handle

Returns
- Status code

int64_t 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

Parameters
- client -[in] The esp_http_client handle

Returns
- (-1) Chunked transfer
- Content-Length value as bytes

esp_err_t esp_http_client_close(esp_http_client_handle_t client)
Close http connection, still kept all http request resources.

Parameters
- client -[in] The esp_http_client handle

Returns
- ESP_OK
- ESP_FAIL

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.

Parameters
- client -[in] The esp_http_client handle

Returns
- ESP_OK
- ESP_FAIL
**esp_http_client_transport_t**

*esp_http_client_get_transport_type*(esp_http_client_handle_t client)

Get transport type.

**Parameters**
- **client** - [in] The esp_http_client handle

**Returns**
- HTTP_TRANSPORT_UNKNOWN
- HTTP_TRANSPORT_OVER_TCP
- HTTP_TRANSPORT_OVER_SSL

**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. When `disable_auto_redirect` is set, the client will not call this function but the event `HTTP_EVENT_REDIRECT` will be dispatched giving the user control over the redirection event.

**Parameters**
- **client** - [in] The esp_http_client handle

**Returns**
- ESP_OK
- ESP_FAIL

**esp_err_t**

*esp_http_client_set_auth_data*(esp_http_client_handle_t client, const char* auth_data, int len)

On receiving a custom authentication header, this API can be invoked to set the authentication information from the header. This API can be called from the event handler.

**Parameters**
- **client** - [in] The esp_http_client handle
- **auth_data** - [in] The authentication data received in the header
- **len** - [in] length of auth_data.

**Returns**
- ESP_ERR_INVALID_ARG
- ESP_OK

**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**
- **client** - [in] 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.

**Parameters**
- **client** - [in] The esp_http_client handle

**Returns**
- true
- false

**int**

*esp_http_client_read_response*(esp_http_client_handle_t client, char* buffer, int len)

Helper API to read larger data chunks. This is a helper API which internally calls `esp_http_client_read` multiple times till the end of data is reached or till the buffer gets full.

**Parameters**
- **client** - [in] The esp_http_client handle
- **buffer** - The buffer
- **len** - [in] The buffer length

**Returns**
- Length of data was read
**esp_err_t esp_http_client_flush_response(esp_http_client_handle_t client, int *len)**

Process all remaining response data. This uses an internal buffer to repeatedly receive, parse, and discard response data until complete data is processed. As no additional user-supplied buffer is required, this may be preferable to `esp_http_client_read_response` in situations where the content of the response may be ignored.

**Parameters**

- **client** [in] The `esp_http_client` handle
- **len** [in] Length of data discarded

**Returns**

- ESP_OK If successful, len will have discarded length
- ESP_FAIL If failed to read response
- ESP_ERR_INVALID_ARG If the client is NULL

**esp_err_t esp_http_client_get_url(esp_http_client_handle_t client, char *url, const int len)**

Get URL from client.

**Parameters**

- **client** [in] The `esp_http_client` handle
- **url** [inout] The buffer to store URL
- **len** [in] The buffer length

**Returns**

- ESP_OK
- ESP_FAIL

**esp_err_t esp_http_client_get_chunk_length(esp_http_client_handle_t client, int *len)**

Get Chunk-Length from client.

**Parameters**

- **client** [in] The `esp_http_client` handle
- **len** [out] Variable to store length

**Returns**

- ESP_OK If successful, len will have length of current chunk
- ESP_FAIL If the server is not a chunked server
- ESP_ERR_INVALID_ARG If the client or len are NULL

**Structures**

```c
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_data

char *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_on_data
    Argument structure for HTTP_EVENT_ON_DATA event.

Public Members

esp_http_client_handle_t client
    Client handle

int64_t data_process
    Total data processed

struct esp_http_client_redirect_event_data
    Argument structure for HTTP_EVENT_REDIRECT event.

Public Members

esp_http_client_handle_t client
    Client handle

int status_code
    Status Code

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

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
SSL client certification, PEM format as string, if the server requires to verify client

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
SSL client key, PEM format as string, if the server requires to verify client

size_t client_key_len
Length of the buffer pointed to by client_key_pem. May be 0 for null-terminated pem

const char *client_key_password
Client key decryption password string

size_t client_key_password_len
String length of the password pointed to by client_key_password

*esp_http_client_proto_ver_t* tls_version
TLS protocol version of the connection, e.g., TLS 1.2, TLS 1.3 (default - no preference)

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 number of redirections on receiving HTTP redirect status code, using default value if zero

int max_authorization_retries
    Max connection retries on receiving HTTP unauthorized status code, using default value if zero. Disables
    authorization retry if -1

http_event_handle_cb event_handler
    HTTP Event Handle

esp_http_client_transport_t transport_type
    HTTP transport type, see esp_http_client_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

const char *common_name
    Pointer to the string containing server certificate common name. If non-NULL, server certificate CN
    must match this name, If NULL, server certificate CN must match hostname.

esp_err_t (*crt_bundle_attach)(void *conf)
    Function pointer to esp_crt_bundle_attach. Enables the use of certification bundle for server verification,
    must be enabled in menuconfig

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

struct ifreq *if_name
    The name of interface for data to go through. Use the default interface without setting

void *ds_data
    Pointer for digital signature peripheral context, see ESP-TLS Documentation for more details

Macros

#define DEFAULT_HTTP_BUF_SIZE

#define ESP_ERR_HTTP_BASE
    Starting number of HTTP error codes

#define ESP_ERR_HTTP_MAX_REDIRECT
    The error exceeds the number of HTTP redirects

#define ESP_ERR_HTTP_CONNECT
    Error open the HTTP connection

#define ESP_ERR_HTTP_WRITE_DATA
    Error write HTTP data

#define ESP_ERR_HTTP_FETCH_HEADER
    Error read HTTP header from server

#define ESP_ERR_HTTP_INVALID_TRANSPORT
    There are no transport support for the input scheme

#define ESP_ERR_HTTP_CONNECTING
    HTTP connection hasn’t been established yet

#define ESP_ERR_HTTP_EAGAIN
    Mapping of errno EAGAIN to esp_err_t

#define ESP_ERR_HTTP_CONNECTION_CLOSED
    Read FIN from peer and the connection closed

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 struct esp_http_client_on_data esp_http_client_on_data_t
    Argument structure for HTTP_EVENT_ON_DATA event.

typedef struct esp_http_client_redirect_event_data esp_http_client_redirect_event_data_t
    Argument structure for HTTP_EVENT_REDIRECT event.

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:
    
    enumerator HTTP_EVENT_ERROR
        This event occurs when there are any errors during execution

    enumerator HTTP_EVENT_ON_CONNECTED
        Once the HTTP has been connected to the server, no data exchange has been performed

    enumerator HTTP_EVENT_HEADERS_SENT
        After sending all the headers to the server

    enumerator HTTP_EVENT_HEADER_SENT
        This header has been kept for backward compatibility and will be deprecated in future versions esp-idf

    enumerator HTTP_EVENT_ON_HEADER
        Occurs when receiving each header sent from the server

    enumerator HTTP_EVENT_ON_DATA
        Occurs when receiving data from the server, possibly multiple portions of the packet

    enumerator HTTP_EVENT_ON_FINISH
        Occurs when finish a HTTP session

    enumerator HTTP_EVENT_DISCONNECTED
        The connection has been disconnected

    enumerator HTTP_EVENT_REDIRECT
        Intercepting HTTP redirects to handle them manually

enum esp_http_client_transport_t
    HTTP Client transport.
    Values:
enumerator HTTP_TRANSPORT_UNKNOWN
Unknown

counter HTTP_TRANSPORT_OVER_TCP
Transport over tcp

counter HTTP_TRANSPORT_OVER_SSL
Transport over ssl

counter esp_http_client_proto_ver_t

Values:

counter ESP_HTTP_CLIENT_TLS_VER_ANY

counter ESP_HTTP_CLIENT_TLS_VER_TLS_1_2

counter ESP_HTTP_CLIENT_TLS_VER_TLS_1_3

counter ESP_HTTP_CLIENT_TLS_VER_MAX

counter esp_http_client_method_t

HTTP method.

Values:

counter HTTP_METHOD_GET

HTTP GET Method

counter HTTP_METHOD_POST

HTTP POST Method

counter HTTP_METHOD_PUT

HTTP PUT Method

counter HTTP_METHOD_PATCH

HTTP PATCH Method

counter HTTP_METHOD_DELETE

HTTP DELETE Method

counter HTTP_METHOD_HEAD

HTTP HEAD Method

counter HTTP_METHOD_NOTIFY

HTTP NOTIFY Method

counter HTTP_METHOD_SUBSCRIBE

HTTP SUBSCRIBE Method
enumerator HTTP_METHOD_UNSUBSCRIBE
   HTTP UNSUBSCRIBE Method

enumerator HTTP_METHOD_OPTIONS
   HTTP OPTIONS Method

enumerator HTTP_METHOD_COPY
   HTTP COPY Method

enumerator HTTP_METHOD_MOVE
   HTTP MOVE Method

enumerator HTTP_METHOD_LOCK
   HTTP LOCK Method

enumerator HTTP_METHOD_UNLOCK
   HTTP UNLOCK Method

enumerator HTTP_METHOD_PROPFIND
   HTTP PROPFIND Method

enumerator HTTP_METHOD_PROPPATCH
   HTTP PROPPATCH Method

enumerator HTTP_METHOD_MKCOL
   HTTP MKCOL Method

enumerator HTTP_METHOD_MAX

enum esp_http_client_auth_type_t
   HTTP Authentication type.
   Values:

   enumerator HTTP_AUTH_TYPE_NONE
      No authention

   enumerator HTTP_AUTH_TYPE_BASIC
      HTTP Basic authentication

   enumerator HTTP_AUTH_TYPE_DIGEST
      HTTP Disgest authentication

enum HttpStatusCode
   Enum for the HTTP status codes.
   Values:

   enumerator HttpStatus_Ok
enumerator HttpStatus_MultipleChoices
enumerator HttpStatus_MovedPermanently
enumerator HttpStatus_Found
enumerator HttpStatus_SeeOther
enumerator HttpStatus_TemporaryRedirect
enumerator HttpStatus_PermanentRedirect
enumerator HttpStatus_BadRequest
enumerator HttpStatus_Unauthorized
enumerator HttpStatus_Forbidden
enumerator HttpStatus_NotFound
enumerator HttpStatus_InternalError

2.2.6 ESP Local Control

Overview

ESP Local Control (esp_local_ctrl) component in ESP-IDF provides capability to control an ESP device over HTTPS or Bluetooth® Low Energy. 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 Bluetooth Low Energy transport is performed as follows:

```c
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
      }
    },
    .proto_sec = {
      .version = PROTOCOL_SEC0,
      .custom_handle = NULL,
      .sec_params = NULL,
    },
  },
};
```
Chapter 2. API Reference

(continued from previous page)

```c
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:

```c
/* Set the configuration */
httpd_ssl_config_t https_conf = HTTPD_SSL_CONFIG_DEFAULT();

/* Load server certificate */
extern const unsigned char servercert_start[] asm("_binary_servercert_pem_
    \_start");
extern const unsigned char servercert_end[] asm("_binary_servercert_pem_
    \_end");
https_conf.servercert = servercert_start;
https_conf.servercert_len = servercert_end - servercert_start;

/* Load server private key */
extern const unsigned char prvtkey_pem_start[] asm("_binary_prvtkey_pem_
    \_start");
extern const unsigned char prvtkey_pem_end[] asm("_binary_prvtkey_pem_
    \_end");
https_conf.prvtkey_pem = prvtkey_pem_start;
https_conf.prvtkey_len = prvtkey_pem_end - prvtkey_pem_start;

esp_local_ctrl_config_t config = {
    .transport = ESP_LOCAL_CTRL_TRANSPORT_HTTPD,
    .transport_config = {
        .httpd = &https_conf
    },
    .proto_sec = {
        .version = PROTOCOM_SEC0,
        .custom_handle = NULL,
        .sec_params = NULL
    },
    .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));
```

You may set security for transport in ESP local control using following options:
1. PROTOCOM_SEC2: specifies that SRP6a-based key exchange and end-to-end encryption based on AES-GCM are used. This is the most preferred option as it adds a robust security with Augmented PAKE protocol, i.e., SRP6a.
2. PROTOCOM_SEC1: specifies that Curve25519-based key exchange and end-to-end encryption based on AES-CTR are used.
3. PROTOCOM_SEC0: specifies that data will be exchanged as a plain text (no security).
4. PROTOCOM_SEC_CUSTOM: you can define your own security requirement. Please note that you will also have to provide custom_handle of type protocomm_security_t * in this context.

Note: The respective security schemes need to be enabled through the project configuration menu. Please refer to the Enabling protocom security version section in Protocol Communication for more details.

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 it is a string or bytestream). For data types with fixed-length property value, 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 up to the application, hence they may be used as enumerations, bitfields, 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.

```c
/* 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.

```c
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_,
    void *usr_ctx)
{
    for (uint32_t i = 0; i < props_count; i++) {
        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 *free_fn */
        }
    }
}
```
Here is an example of set_prop_values() handler. Notice how we restrict from writing to read-only properties.

```c
static esp_err_t set_property_values(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 establishes 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 translates 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 1: Endpoints provided by ESP Local Control

<table>
<thead>
<tr>
<th>Endpoint Name <em>(Bluetooth Low Energy + GATT Server)</em></th>
<th>URI <em>(HTTPS Server + mDNS)</em></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp_local_ctrl/version</td>
<td>https://&lt;mdns-hostname&gt;.local/esp_local_ctrl/version</td>
<td>Endpoint used for retrieving version string</td>
</tr>
<tr>
<td>esp_local_ctrl/control</td>
<td>https://&lt;mdns-hostname&gt;.local/esp_local_ctrl/control</td>
<td>Endpoint used for sending or receiving control messages</td>
</tr>
</tbody>
</table>

### API Reference

**Header File**

- components/esp_local_ctrl/include/esp_local_ctrl.h
- This header file can be included with:

  ```
  #include "esp_local_ctrl.h"
  ```

- This header file is a part of the API provided by the esp_local_ctrl component. To declare that your component depends on esp_local_ctrl, add the following to your CMakeLists.txt:

  ```
  REQUIRE{{--}}s esp_local_ctrl
  ```
  or

  ```
  PRIV_REQUIRE{{--}}s esp_local_ctrl
  ```

**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.

  **Parameters** `config` – [in] Pointer to configuration structure

  **Returns**
  - ESP_OK : Success
  - ESP_FAIL : Failure

- `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`

  **Parameters** `prop` – [in] Property description structure

  **Returns**
  - ESP_OK : Success
  - ESP_FAIL : Failure
**Chapter 2. API Reference**

`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.

**Parameters**

- `name` - [in] Name of the property to remove

**Returns**

- ESP_OK: Success
- ESP_ERR_NOT_FOUND: Failure

`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.

**Parameters**

- `name` - [in] Name of the property to find

**Returns**

- Pointer to property
- NULL if not found

`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.

**Parameters**

- `ep_name` - [in] Name of the endpoint
- `handler` - [in] Endpoint handler function
- `user_ctx` - [in] User data

**Returns**

- ESP_OK: Success
- ESP_FAIL: Failure

---

**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()`.

---

**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**

```c
char *name
```

Unique name of property

```c
uint32_t type
```

Type of property. This may be set to application defined enums

```c
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

```c
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

```c
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.

```c
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**

```c
void *data
```

Pointer to memory holding property value

```c
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

Param props_count [in] Total elements in the props array
Param props [in] Array of properties, the current values for which have been requested by the client
Param prop_values [out] Array of empty property values, the elements of which need to be populated with the current values of those properties specified by props argument
Param usr_ctx [in] This provides value of the usr_ctx field of esp_local_ctrl_handlers_t structure

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

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.

Param props_count [in] Total elements in the props array
Param props [in] Array of properties, the values for which the client requests to change
Param prop_values [in] Array of property values, the elements of which need to be used for updating those properties specified by props argument
Param usr_ctx [in] This provides value of the usr_ctx field of esp_local_ctrl_handlers_t structure

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
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_proto_sec_cfg
Protocom security configs

Public Members

esp_local_ctrl_proto_sec_t version
This sets protocom security version, sec0/sec1 or custom If custom, user must provide handle via proto_sec_custom_handle below

void *custom_handle
Custom security handle if security is set custom via proto_sec above This handle must follow protocomm_security_t signature

const void *pop
Proof of possession to be used for local control. Could be NULL.

const void *sec_params
Pointer to security params (NULL if not needed). This is not needed for protocomm security 0 This pointer should hold the struct of type esp_local_ctrl_security1_params_t for protocomm security 1 and esp_local_ctrl_security2_params_t for protocomm security 2 respectively. Could be NULL.

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_proto_sec_cfg_t proto_sec
Security version and POP

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
Chapter 2. API Reference

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 CON-
FIG_BT_BLUEDROID_ENABLED and include protocomm_ble.h.

typedef struct httpd_config esp_local_ctrl_transport_config_httpd_t

Configuration for transport mode HTTPD.

This is a forward declaration for httpd_ssl_config_t (for HTTPS) or httpd_config_t (for HTTP)

typedef enum esp_local_ctrl_proto_sec esp_local_ctrl_proto_sec_t

Security types for esp_local_control.

typedef protocomm_security1_params_t esp_local_ctrl_security1_params_t

typedef protocomm_security2_params_t esp_local_ctrl_security2_params_t

typedef struct esp_local_ctrl_proto_sec_cfg esp_local_ctrl_proto_sec_cfg_t

Protocol security configs

typedef struct esp_local_ctrl_config esp_local_ctrl_config_t

Configuration structure to pass to esp_local_ctl_start()
Enumerations

enum esp_local_ctrl_proto_sec
  Security types for esp_local_control.
  Values:
  
  enumerator PROTOCOL_SEC0
  enumerator PROTOCOL_SEC1
  enumerator PROTOCOL_SEC2
  enumerator PROTOCOL_SEC_CUSTOM

2.2.7 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.

Note: The ESP-IDF component esp_serial_slave_link has been moved from ESP-IDF since version v5.0 to a separate repository:

- ESSL component on GitHub

To add ESSL component in your project, please run idf.py add-depending espressif/esp_serial_slave_link.

Espressif Device Protocols

For more details about Espressif device protocols, see the following documents.

ESP SPI Slave HD (Half Duplex) Mode Protocol

<table>
<thead>
<tr>
<th></th>
<th>ESP32-S2</th>
<th>ESP32-S3</th>
<th>ESP32-C2</th>
<th>ESP32-C6</th>
<th>ESP32-H2</th>
<th>ESP32-P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI Slave HD</td>
<td>N</td>
<td>Y (v2)</td>
<td>Y (v2)</td>
<td>Y (v2)</td>
<td>Y (v2)</td>
<td>Y (v2)</td>
</tr>
<tr>
<td>Tohost intr</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Frhost intr</td>
<td>2 *</td>
<td>2 *</td>
<td>2 *</td>
<td>2 *</td>
<td>2 *</td>
<td>2 *</td>
</tr>
<tr>
<td>TX DMA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>RX DMA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Shared registers</td>
<td>72</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
**Chapter 2. API Reference**

**Introduction**  In the half-duplex mode, the master has to use the protocol defined by the slave to communicate with the slave. Each transaction may consist of the following phases (listed by the order they should exist):

- Command: 8-bit, master to slave
  This phase determines the rest phases of the transactions. See [Supported Commands](#).
- Address: 8-bit, master to slave, optional
  For some commands (WRBUF, RDBUF), this phase specifies the address of the shared register to write to/read from. For other commands with this phase, they are meaningless but still have to exist in the transaction.
- Dummy: 8-bit, floating, optional
  This phase is the turnaround time between the master and the slave on the bus, and also provides enough time for the slave to prepare the data to send to the master.
- Data: variable length, the direction is also determined by the command.
  This may be a data OUT phase, in which the direction is slave to master, or a data IN phase, in which the direction is master to slave.

The **direction** means which side (master or slave) controls the MOSI, MISO, WP, and HD pins.

**Data IO Modes**  In some IO modes, more data wires can be used to send the data. As a result, the SPI clock cycles required for the same amount of data will be less than in the 1-bit mode. For example, in QIO mode, address and data (IN and OUT) should be sent on all 4 data wires (MOSI, MISO, WP, and HD). Here are the modes supported by the ESP32-S2 SPI slave and the wire number (WN) used in corresponding modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Command WN</th>
<th>Address WN</th>
<th>Dummy cycles</th>
<th>Data WN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DOUT</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>DIO</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>QOUT</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>QIO</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>QPI</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Normally, which mode is used is determined by the command sent by the master (See [Supported Commands](#)), except the QPI mode.

**QPI Mode**  The QPI mode is a special state of the SPI Slave. The master can send the ENQPI command to put the slave into the QPI mode state. In the QPI mode, the command is also sent in 4-bit, thus it is not compatible with the normal modes. The master should only send QPI commands when the slave is in QPI mode. To exit from the QPI mode, master can send the EXQPI command.

**Supported Commands**

**Note:** The command name is in a master-oriented direction. For example, WRBUF means master writes the buffer of slave.
### API Reference

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Command</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRBUF</td>
<td>Write buffer</td>
<td>0x01</td>
<td>Buf addr</td>
<td>master to slave, no longer than buffer size</td>
</tr>
<tr>
<td>RDBUF</td>
<td>Read buffer</td>
<td>0x02</td>
<td>Buf addr</td>
<td>slave to master, no longer than buffer size</td>
</tr>
<tr>
<td>WRDMA</td>
<td>Write DMA</td>
<td>0x03</td>
<td>8 bits</td>
<td>master to slave, no longer than length provided by slave</td>
</tr>
<tr>
<td>RDDMA</td>
<td>Read DMA</td>
<td>0x04</td>
<td>8 bits</td>
<td>slave to master, no longer than length provided by slave</td>
</tr>
<tr>
<td>SEG_DONE</td>
<td>Segments done</td>
<td>0x05</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>ENQPI</td>
<td>Enter QPI mode</td>
<td>0x06</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>WR_DONE</td>
<td>Write segments done</td>
<td>0x07</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CMD8</td>
<td>Interrupt</td>
<td>0x08</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CMD9</td>
<td>Interrupt</td>
<td>0x09</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>CMDA</td>
<td>Interrupt</td>
<td>0x0A</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>EXQPI</td>
<td>Exit QPI mode</td>
<td>0xDD</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Moreover, WRBUF, RDBUF, WRDMA, and RDDMA commands have their 2-bit and 4-bit version. To do transactions in 2-bit or 4-bit mode, send the original command ORed by the corresponding command mask below. For example, command 0xA1 means WRBUF in QIO mode.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-bit</td>
<td>0x00</td>
</tr>
<tr>
<td>DOUT</td>
<td>0x10</td>
</tr>
<tr>
<td>DIO</td>
<td>0x50</td>
</tr>
<tr>
<td>QOUT</td>
<td>0x20</td>
</tr>
<tr>
<td>QIO</td>
<td>0xA0</td>
</tr>
<tr>
<td>QPI</td>
<td>0xA0</td>
</tr>
</tbody>
</table>

**Segment Transaction Mode** Segment transaction mode is the only mode supported by the SPI Slave HD driver for now. In this mode, for a transaction the slave loads onto the DMA, the master is allowed to read or write in segments. In this way, the master does not have to prepare a large buffer as the size of data provided by the slave. After the master finishes reading/writing a buffer, it has to send the corresponding termination command to the slave as a synchronization signal. The slave driver will update new data (if exist) onto the DMA upon seeing the termination command.

The termination command is WR_DONE (0x07) for WRDMA and CMD8 (0x08) for RDDMA.

Here is an example for the flow the master read data from the slave DMA:

1. The slave loads 4092 bytes of data onto the RDDMA.
2. The master do seven RDDMA transactions, each of them is 512 bytes long, and reads the first 3584 bytes from the slave.
3. The master do the last RDDMA transaction of 512 bytes (equal, longer, or shorter than the total length loaded by the slave are all allowed). The first 508 bytes are valid data from the slave, while the last 4 bytes are meaningless bytes.
4. The master sends CMD8 to the slave.
5. The slave loads another 4092 bytes of data onto the RDDMA.
6. The master can start new reading transactions after it sends the CMD8.

**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. (optional)
2. Frhost Interrupts: The master can inform the slave about certain events.
3. TX FIFO (master to slave): The slave can receive data from the master in units of receiving buffers.
   - 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 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 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.

The services provided by the slave depends on the slave’s model. See *SPI Slave Capabilities of Espressif Chips* for more details.

**Initialization of ESP Serial Slave Link**

**ESP SDIO Slave**

The ESP SDIO slave link (ESSL SDIO) devices relies on the SDMMC component. It includes the usage of communicating with ESP SDIO Slave device via the SDMMC Host or SDSPI Host feature. 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.

---

**ESP SPI Slave**

**Note:** If you are communicating with the ESP SDIO Slave device through SPI interface, you should use the SDIO interface instead.

Has not been supported yet.

---

**APIs**

After the initialization process above is performed, you can call the APIs below to make use of the services provided by the slave:

**Tohost Interrupts (Optional)**

1. Call `essl_get_intr_ena()` to know which events trigger the interrupts to the master.
2. Call `essl_set_intr_ena()` to set the events that 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.

**Fhost Interrupts**

1. 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 tries 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 updates 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-S2 SDIO host and slave communicate with each other. The host uses the ESSL SDIO:

peripherals/sdio

Please refer to the specific example README.md for details.
Chapter 2. API Reference

API Reference

Header File

- components/driver/test_apps/components/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.

**Parameters**
- **handle** – Handle of an ESSL device.
- **wait_ms** – Millisecond to wait before timeout, will not wait at all if set to 0-9.

**Returns**
- **ESP_OK**: If success
- **ESP_ERR_NOT_SUPPORTED**: Current device does not support this function.
- Other value returned from lower layer `init`.

**esp_err_t essl_wait_for_ready (essl_handle_t handle, uint32_t wait_ms)**

Wait for interrupt of an ESSL slave device.

**Parameters**
- **handle** – Handle of an ESSL device.
- **wait_ms** – Millisecond to wait before timeout, will not wait at all if set to 0-9.

**Returns**
- **ESP_OK**: If success
- **ESP_ERR_NOT_SUPPORTED**: Current device does not support this function.
- One of the error codes from SDMMC host controller

**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`.

**Parameters**
- **handle** – Handle of an ESSL device.
- **out_tx_num** – Output of buffer num that host can send data to ESSL slave.
- **wait_ms** – Millisecond to wait before timeout, will not wait at all if set to 0-9.

**Returns**
- **ESP_OK**: Success
- **ESP_ERR_NOT_SUPPORTED**: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller

**esp_err_t essl_get_rx_data_size (essl_handle_t handle, uint32_t *out_rx_size, uint32_t wait_ms)**

Get the size, in bytes, of the data that the ESSL slave is ready to send

**Parameters**
- **handle** – Handle of an ESSL device.
- **out_rx_size** – Output of data size to read from slave, in bytes
- **wait_ms** – Millisecond to wait before timeout, will not wait at all if set to 0-9.

**Returns**
- **ESP_OK**: Success
- **ESP_ERR_NOT_SUPPORTED**: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller

**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 an ESSL device.

**Returns**
- **ESP_OK**: Success
- **ESP_ERR_NOT_SUPPORTED**: This API is not supported in this mode
- **ESP_ERR_INVALID_ARG**: Invalid argument, handle is not init.
**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 ESSL Slave. The Slave receives the packet into buffers whose size is `buffer_size` (configured during initialization).

**Parameters**
- `handle` - Handle of an 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.

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG: Invalid argument, handle is not init or other argument is not valid.
- ESP_ERR_TIMEOUT: No buffer to use, or error from SDMMC host controller.
- ESP_ERR_NOT_FOUND: Slave is not ready for receiving.
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller.

**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 ESSL slave.

**Parameters**
- `handle` - Handle of an ESSL device.
- `out_data` - [out] 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_length` - [out] 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.

**Returns**
- ESP_OK Success: All the data has been read from the slave.
- ESP_ERR_INVALID_ARG: Invalid argument, The handle is not initialized or the other arguments are invalid.
- ESP_ERR_NOT_FOUND: Read was successful, but there is still data remaining.
- ESP_ERR_NOT_SUPPORTED: Slave is not ready to send data.
- ESP_ERR_NOT_SUPPORTED: This API is not supported in this mode
- One of the error codes from SDMMC/SPI host controller.

**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 ESSL slave.

**Parameters**
- `handle` - Handle of an ESSL device.
- `addr` - Address of register to write. For SDIO, valid address: 0-59. For SPI, see `essl_spi.h`
- `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.

**Returns**
- ESP_OK Success
- One of the error codes from SDMMC/SPI host controller

**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 ESSL slave.

---

**Note:** sdio 28-31 are reserved, the lower API helps to skip.
Parameters
  • handle – Handle of a \texttt{essl} device.
  • addr – Address of register to read. For SDIO, Valid address: 0-27, 32-63 (28-31 reserved, return interrupt bits on read). For SPI, see \texttt{essl\_spi.h}
  • 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.

Returns
  • ESP\_OK Success
  • One of the error codes from SDMMC/SPI host controller

\texttt{esp\_err\_t essl\_wait\_int (essl\_handle\_t handle, uint32\_t wait\_ms)}

wait for an interrupt of the slave

Parameters
  • handle – Handle of an ESSL device.
  • wait\_ms – Millisecond to wait before timeout, will not wait at all if set to 0-9.

Returns
  • ESP\_OK: If interrupt is triggered.
  • ESP\_ERR\_NOT\_SUPPORTED: Current device does not support this function.
  • ESP\_ERR\_TIMEOUT: No interrupts before timeout.

\texttt{esp\_err\_t essl\_clear\_intr (essl\_handle\_t handle, uint32\_t intr\_mask, uint32\_t wait\_ms)}

Clear interrupt bits of ESSL slave. All the bits set in the mask will be cleared, while other bits will stay the same.

Parameters
  • handle – Handle of an 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.

Returns
  • ESP\_OK: Success
  • ESP\_ERR\_NOT\_SUPPORTED: Current device does not support this function.
  • One of the error codes from SDMMC host controller

\texttt{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 ESSL slave.

Parameters
  • handle – Handle of an 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.

Returns
  • ESP\_OK: Success
  • ESP\_ERR\_NOT\_SUPPORTED: Current device does not support this function.
  • One of the error codes from SDMMC host controller

\texttt{esp\_err\_t essl\_set\_intr\_ena (essl\_handle\_t handle, uint32\_t ena\_mask, uint32\_t wait\_ms)}

Set interrupt enable bits of ESSL slave. The slave only sends interrupt on the line when there is a bit both the raw status and the enable are set.

Parameters
  • handle – Handle of an 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.

Returns
  • ESP\_OK: Success
  • ESP\_ERR\_NOT\_SUPPORTED: Current device does not support this function.
  • One of the error codes from SDMMC host controller
**Chapter 2. API Reference**

*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 ESSL slave.

**Parameters**
- `handle` - Handle of an 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.

**Returns**
- ESP_OK Success
- One of the error codes from SDMMC host controller

*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.

**Parameters**
- `handle` - Handle of an 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.

**Returns**
- ESP_OK: Success
- ESP_ERR_NOT_SUPPORTED: Current device does not support this function.
- One of the error codes from SDMMC host controller

**Type Definitions**

typedef struct essl_dev_t *essl_handle_t

Handle of an ESSL device.

**Header File**
- components/driver/test_apps/components/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.

**Parameters**
- `out_handle` - Output of the handle.
- `config` - Configuration for the ESSL SDIO device.

**Returns**
- ESP_OK: on success
- ESP_ERR_NO_MEM: memory exhausted.

*esp_err_t* `essl_sdio_deinit_dev(essl_handle_t handle)`

Deinitialize and free the space used by the ESSL SDIO device.

**Parameters**
- `handle` - Handle of the ESSL SDIO device to deinit.

**Returns**
- ESP_OK: on success
- ESP_ERR_INVALID_ARG: wrong handle passed

**Structures**

struct essl_sdio_config_t

Configuration for the ESSL SDIO device.
Public Members

dsmmc_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.

Header File

- components/driver/test_apps/components/esp_serial_slave_link/include/esp_serial_slave_link/essl_spi.h

Functions

`esp_err_t essl_spi_init_dev (essl_handle_t *out_handle, const essl_spi_config_t *init_config)`

Initialize the ESSL SPI device function list and get its handle.

Parameters

- `out_handle` - [out] Output of the handle
- `init_config` - Configuration for the ESSL SPI device

Returns

- ESP_OK: On success
- ESP_ERR_NO_MEM: Memory exhausted
- ESP_ERR_INVALID_STATE: SPI driver is not initialized
- ESP_ERR_INVALID_ARG: Wrong register ID

`esp_err_t essl_spi_deinit_dev (essl_handle_t handle)`

Deinitialize the ESSL SPI device and free the memory used by the device.

Parameters

- `handle` - Handle of the ESSL SPI device

Returns

- ESP_OK: On success
- ESP_ERR_INVALID_STATE: ESSL SPI is not in use

`esp_err_t essl_spi_read_reg (void *arg, uint8_t addr, uint8_t *out_value, uint32_t wait_ms)`

Read from the shared registers.

Parameters

- `arg` - Context of the component. (Member `arg` from `essl_handle_t`)
- `addr` - Address of the shared registers. (Valid: 0 ~ SOC_SPI_MAXIMUM_BUFFER_SIZE, registers for M/S sync are reserved, see note1).
- `out_value` - [out] Read buffer for the shared registers.
- `wait_ms` - Time to wait before timeout (reserved for future use, user should set this to 0).

Returns

- ESP_OK: success
- ESP_ERR_INVALID_STATE: ESSL SPI has not been initialized.
- ESP_ERR_INVALID_ARG: The address argument is not valid. See note 1.

- or other return value from `esp_spi_device_transmit`.

`esp_err_t essl_spi_get_packet (void *arg, void *out_data, size_t size, uint32_t wait_ms)`

Get a packet from Slave.

Note: The registers for Master/Slave synchronization are reserved. Do not use them. (see `rx sync reg` in `essl_spi_config_t`)
### Parameters
- **arg** – Context of the component. (Member `arg` from `essl_handle_t`)
- **out_data** – [out] Output data address
- **size** – The size of the output data.
- **wait_ms** – Time to wait before timeout (reserved for future use, user should set this to 0).

### Returns
- **ESP_OK**: On Success
- **ESP_ERR_INVALID_STATE**: ESSL SPI has not been initialized.
- **ESP_ERR_INVALID_ARG**: The output data address is neither DMA capable nor 4 byte-aligned
- **ESP_ERR_INVALID_SIZE**: Master requires `size` bytes of data but Slave did not load enough bytes.

```
esp_err_t essl_spi_write_reg(void *arg, uint8_t addr, uint8_t value, uint8_t *out_value, uint32_t wait_ms)
```

Write to the shared registers.

---

**Note:** The registers for Master/Slave synchronization are reserved. Do not use them. (see `tx_sync_reg` in `essl_spi_config_t`)

---

**Note:** Feature of checking the actual written value (`out_value`) is not supported.

### Parameters
- **addr** – Address of the shared registers. (Valid: 0 ~ SOC_SPI_MAXIMUM_BUFFER_SIZE, registers for M/S sync are reserved, see note1)
- **value** – Buffer for data to send, should be align to 4.
- **out_value** – [out] Not supported, should be set to NULL.
- **wait_ms** – Time to wait before timeout (reserved for future use, user should set this to 0).

### Returns
- **ESP_OK**: success
- **ESP_ERR_INVALID_STATE**: ESSL SPI has not been initialized.
- **ESP_ERR_INVALID_ARG**: The address argument is not valid. See note 1.
- **ESP_ERR_NOT_SUPPORTED**: Should set `out_value` to NULL. See note 2.
- **or other return value from :cpp:func:`spi_device_transmit`.

```
esp_err_t essl_spi_send_packet(void *arg, const void *data, size_t size, uint32_t wait_ms)
```

Send a packet to Slave.

### Parameters
- **arg** – Context of the component. (Member `arg` from `essl_handle_t`)
- **data** – Address of the data to send
- **size** – Size of the data to send.
- **wait_ms** – Time to wait before timeout (reserved for future use, user should set this to 0).

### Returns
- **ESP_OK**: On success
- **ESP_ERR_INVALID_STATE**: ESSL SPI has not been initialized.
- **ESP_ERR_INVALID_ARG**: The data address is not DMA capable
- **ESP_ERR_INVALID_SIZE**: Master will send `size` bytes of data but Slave did not load enough RX buffer
void `essl_spi_reset_cnt` (void *arg)

Reset the counter in Master context.

**Note:** Shall only be called if the slave has reset its counter. Else, Slave and Master would be desynchronized.

**Parameters**
- `arg` - Context of the component. (Member `arg` from `essl_handle_t`)

`esp_err_t ` `essl_spi_rdbuf` `(` `spi_device_handle_t` `spi`, uint8_t *out_data, int addr, int len, uint32_t flags `)`

Read the shared buffer from the slave in ISR way.

**Note:** The slave’s HW doesn’t guarantee the data in one SPI transaction is consistent. It sends data in unit of byte. In other words, if the slave SW attempts to update the shared register when a `rdbuf` SPI transaction is in-flight, the data got by the master will be the combination of bytes of different writes of slave SW.

**Parameters**
- `spi` - SPI device handle representing the slave
- `out_data` - [out] Buffer for read data, strongly suggested to be in the DRAM and aligned to 4
- `addr` - Address of the slave shared buffer
- `len` - Length to read
- `flags` - SPI_TRANS_* flags to control the transaction mode of the transaction to send.

**Returns**
- ESP_OK: on success
- or other return value from `cpp:func:spl_device_transmit`.

`esp_err_t ` `essl_spi_rdbuf_polling` `(` `spi_device_handle_t` `spi`, uint8_t *out_data, int addr, int len, uint32_t flags `)`

Read the shared buffer from the slave in polling way.

**Note:** out_data should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the len is shorter than a word.

**Parameters**
- `spi` - SPI device handle representing the slave
- `out_data` - [out] Buffer for read data, strongly suggested to be in the DRAM and aligned to 4
- `addr` - Address of the slave shared buffer
- `len` - Length to read
- `flags` - SPI_TRANS_* flags to control the transaction mode of the transaction to send.

**Returns**
- ESP_OK: on success
- or other return value from `cpp:func:spl_device_transmit`.

`esp_err_t ` `essl_spi_wrbuf` `(` `spi_device_handle_t` `spi`, const uint8_t *data, int addr, int len, uint32_t flags `)`

Write the shared buffer of the slave in ISR way.
Note: `out_data` should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the `len` is shorter than a word.

### Parameters
- **spi** – SPI device handle representing the slave
- **data** – Buffer for data to send, strongly suggested to be in the DRAM
- **addr** – Address of the slave shared buffer,
- **len** – Length to write
- **flags** – SPI_TRANS_* flags to control the transaction mode of the transaction to send.

### Returns
- ESP_OK: success
- or other return value from `cpp:func:spi_device_transmit`.

```c
esp_err_t essl_spi_wrbuf_polling(spi_device_handle_t spi, const uint8_t*data, int addr, int len, uint32_t flags)
```

Write the shared buffer of the slave in polling way.

Note: `out_data` should be prepared in words and in the DRAM. The buffer may be written in words by the DMA. When a byte is written, the remaining bytes in the same word will also be overwritten, even the `len` is shorter than a word.

### Parameters
- **spi** – SPI device handle representing the slave
- **data** – Buffer for data to send, strongly suggested to be in the DRAM
- **addr** – Address of the slave shared buffer,
- **len** – Length to write
- **flags** – SPI_TRANS_* flags to control the transaction mode of the transaction to send.

### Returns
- ESP_OK: success
- or other return value from `cpp:func:spi_device_polling_transmit`.

```c
esp_err_t essl_spi_rddma(spi_device_handle_t spi, uint8_t*out_data, int len, int seg_len, uint32_t flags)
```

Receive long buffer in segments from the slave through its DMA.

Note: This function combines several `cpp:func:essl_spi_rddma_seg` and one `cpp:func:essl_spi_rddma_done` at the end. Used when the slave is working in segment mode.

### Parameters
- **spi** – SPI device handle representing the slave
- **out_data** – [out] Buffer to hold the received data, strongly suggested to be in the DRAM and aligned to 4
- **len** – Total length of data to receive.
- **seg_len** – Length of each segment, which is not larger than the maximum transaction length allowed for the spi device. Suggested to be multiples of 4. When set < 0, means send all data in one segment (the rddma_done will still be sent.)
- **flags** – SPI_TRANS_* flags to control the transaction mode of the transaction to send.

### Returns
- ESP_OK: success
- or other return value from `cpp:func:spi_device_transmit`. 

 Espressif Systems } 168 Release master

Submit Document Feedback
### esp_err_t essl_spi_rddma_seg

```c
estrl_spi_rddma_seg(spi_device_handle_t spi, uint8_t*out_data, int seg_len, uint32_t flags)
```

Read one data segment from the slave through its DMA.

**Note:** To read long buffer, call `cpp:func:essl_spi_rddma` instead.

**Parameters**
- `spi` - SPI device handle representing the slave
- `out_data` - [out] Buffer to hold the received data. Strongly suggested to be in the DRAM and aligned to 4
- `seg_len` - Length of this segment
- `flags` - SPI_TRANS_* flags to control the transaction mode of the transaction to send.

**Returns**
- ESP_OK: success
- or other return value from `cpp:func:spi_device_transmit`.

### esp_err_t essl_spi_rddma_done

```c
estrl_spi_rddma_done(spi_device_handle_t spi, uint32_t flags)
```

Send the `rddma_done` command to the slave. Upon receiving this command, the slave will stop sending the current buffer even there are data unsent, and maybe prepare the next buffer to send.

**Note:** This is required only when the slave is working in segment mode.

**Parameters**
- `spi` - SPI device handle representing the slave
- `flags` - SPI_TRANS_* flags to control the transaction mode of the transaction to send.

**Returns**
- ESP_OK: success
- or other return value from `cpp:func:spi_device_transmit`.

### esp_err_t essl_spi_wrdma

```c
estrl_spi_wrdma(spi_device_handle_t spi, const uint8_t*data, int len, int seg_len, uint32_t flags)
```

Send long buffer in segments to the slave through its DMA.

**Note:** This function combines several `cpp:func:essl_spi_wrdma_seg` and one `cpp:func:essl_spi_wrdma_done` at the end. Used when the slave is working in segment mode.

**Parameters**
- `spi` - SPI device handle representing the slave
- `data` - Buffer for data to send, strongly suggested to be in the DRAM
- `len` - Total length of data to send.
- `seg_len` - Length of each segment, which is not larger than the maximum transaction length allowed for the SPI device. Suggested to be multiples of 4. When set < 0, means send all data in one segment (the `wrdma_done` will still be sent.)
- `flags` - SPI_TRANS_* flags to control the transaction mode of the transaction to send.

**Returns**
- ESP_OK: success
- or other return value from `cpp:func:spi_device_transmit`.

### esp_err_t essl_spi_wrdma_seg

```c
estrl_spi_wrdma_seg(spi_device_handle_t spi, const uint8_t*data, int seg_len, uint32_t flags)
```

Send one data segment to the slave through its DMA.

**Note:** To send long buffer, call `cpp:func:essl_spi_wrdma` instead.
Parameters

- **spi** – SPI device handle representing the slave
- **data** – Buffer for data to send, strongly suggested to be in the DRAM
- **seg_len** – Length of this segment
- **flags** – SPI_TRANS_* flags to control the transaction mode of the transaction to send.

Returns

- ESP_OK: success
- or other return value from :cpp:func:`spi_device_transmit`.

```c
esp_err_t essl_spi_wrdma_done(spi_device_handle_t spi, uint32_t flags)
```

Send the wrdma_done command to the slave. Upon receiving this command, the slave will stop receiving, process the received data, and maybe prepare the next buffer to receive.

**Note:** This is required only when the slave is working in segment mode.

**Structures**

```c
struct essl_spi_config_t
```

Configuration of ESSL SPI device.

**Public Members**

```c
spi_device_handle_t *spi
```

Pointer to SPI device handle.

```c
uint32_t tx_buf_size
```

The pre-negotiated Master TX buffer size used by both the host and the slave.

```c
uint8_t tx_sync_reg
```

The pre-negotiated register ID for Master-TX-SLAVE-RX synchronization. 1 word (4 Bytes) will be reserved for the synchronization.

```c
uint8_t rx_sync_reg
```

The pre-negotiated register ID for Master-RX-Slave-TX synchronization. 1 word (4 Bytes) will be reserved for the synchronization.

### 2.2.8 ESP x509 Certificate Bundle
Overview

The ESP x509 Certificate Bundle API provides an easy way to include a bundle of custom x509 root certificates for TLS server verification.

Note: The bundle is currently not available when using WolfSSL.

The bundle comes with the complete list of root certificates from Mozilla’s NSS root certificate store. Using the gen_crt_bundle.py python utility, the certificates’ subject name and public key are stored in a file and embedded in the ESP32-S2 binary.

When generating the bundle you may choose between:

- The full root certificate bundle from Mozilla, containing more than 130 certificates. The current bundle was updated Tue Aug 22 03:12:04 2023 GMT.
- A pre-selected filter list of the name of the most commonly used root certificates, reducing the amount of certificates to around 41 while still having around 90% absolute usage coverage and 99% market share coverage according to SSL certificate authorities statistics.

In addition, it is possible to specify a path to a certificate file or a directory containing certificates which then will be added to the generated bundle.

Note: Trusting all root certificates means the list will have to be updated if any of the certificates are retracted. This includes removing them from cacrt_all.pem.

Configuration

Most configuration is done through menuconfig. CMake generates the bundle according to the configuration and embed it.

- `CONFIG_MBEDTLS_CERTIFICATE_BUNDLE`: automatically build and attach the bundle.
- `CONFIG_MBEDTLS_DEFAULT_CERTIFICATE_BUNDLE`: decide which certificates to include from the complete root certificate list.
- `CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE_PATH`: specify the path of any additional certificates to embed in the bundle.

To enable the bundle when using ESP-TLS simply pass the function pointer to the bundle attach function:

```c
esp_tls_cfg_t cfg = {
    .crt_bundle_attach = esp_crt_bundle_attach,
};
```

This is done to avoid embedding the certificate bundle unless activated by the user.

If using mbedTLS directly then the bundle may be activated by directly calling the attach function during the setup process:

```c
mbedtls_ssl_config conf;
mbedtls_ssl_config_init(&conf);
esp_crt_bundle_attach(&conf);
```

Generating the List of Root Certificates

The list of root certificates comes from Mozilla’s NSS root certificate store, which can be found here

The list can be downloaded and created by running the script `mk-ca-bundle.pl` that is distributed as a part of curl.
Another alternative would be to download the finished list directly from the curl website: CA certificates extracted from Mozilla

The common certificates bundle were made by selecting the authorities with a market share of more than 1% from w3tech’s SSL Survey.

These authorities were then used to pick the names of the certificates for the filter list, cmn_crt_authorities.csv, from this list provided by Mozilla.

Updating the Certificate Bundle

The bundle is embedded into the app and can be updated along with the app by an OTA update. If you want to include a more up-to-date bundle than the bundle currently included in ESP-IDF, then the certificate list can be downloaded from Mozilla as described in Generating the List of Root Certificates.

Periodic Sync

The bundle is kept updated by periodic sync with the Mozilla’s NSS root certificate store. The deprecated certs from the upstream bundle are added to deprecated list (for compatibility reasons) in ESP-IDF minor or patch release. If required, the deprecated certs can be added to the default bundle by enabling CONFIG_MBEDTLS_CERTIFICATE.Bundle_DEPRECATED_LIST. The deprecated certs shall be removed (reset) on the next major ESP-IDF release.

Application Examples

Simple HTTPS example that uses ESP-TLS to establish a secure socket connection using the certificate bundle with two custom certificates added for verification: protocols/https_x509_bundle.

HTTPS example that uses ESP-TLS and the default bundle: protocols/https_request.

HTTPS example that uses mbedTLS and the default bundle: protocols/https_mbedtls.

API Reference

Header File

- components/mbedtls/esp_crt_bundle/include/esp_crt_bundle.h

This header file can be included with:

```c
#include "esp_crt_bundle.h"
```

- This header file is a part of the API provided by the mbedtls component. To declare that your component depends on mbedtls, add the following to your CMakeLists.txt:

```cmake
REQUIRES mbedtls
```

or

```cmake
PRIV_REQUIRES mbedtls
```

Functions

```c
Esp_err_t esp_crt_bundle_attach (void *conf)
```

Attach and enable use of a bundle for certificate verification.

Attach and enable use of a bundle for certificate verification through a verification callback. If no specific bundle has been set through esp_crt_bundle_set() it will default to the bundle defined in menuconfig and embedded in the binary.

Parameters

- **conf** - [in] The config struct for the SSL connection.
Returns

- ESP_OK if adding certificates was successful.
- Other if an error occurred or an action must be taken by the calling process.

```c
void esp_crt_bundle_detach (mbedtls_ssl_config *conf)
```

Disables and deallocates the certification bundle.

Removes the certificate verification callback and deallocates used resources.

**Parameters**

`conf` - [in] The config struct for the SSL connection.

```c
esp_err_t esp_crt_bundle_set (const uint8_t *x509_bundle, size_t bundle_size)
```

Sets the default certificate bundle used for verification.

Overrides the default certificate bundle only in case of successful initialization. In most use cases the bundle should be set through menuconfig. The bundle needs to be sorted by subject name since binary search is used to find certificates.

**Parameters**

- `x509_bundle` - [in] A pointer to the certificate bundle.

**Returns**

- ESP_OK if adding certificates was successful.
- Other if an error occurred or an action must be taken by the calling process.

### 2.2.9 HTTP Server

**Overview**

The HTTP Server component provides an ability for running a lightweight web server on ESP32-S2. Following are detailed steps to use the API exposed by HTTP Server:

- `httpd_start()`: Creates an instance of HTTP server, allocates 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 closes all open connections, removes registered URI handlers and resets 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**

```c
/* 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, HTTPD_RESP_USE_STRLEN);
    return ESP_OK;
}
```

(continues on next page)
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 content[100];

  /* 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;
  }

  /* Send a simple response */
  const char resp[] = "URI POST Response";
  httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
  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.

**Websocket Server**

The HTTP server component provides websocket support. The websocket feature can be enabled in menuconfig using the `CONFIG_HTTPD_WS_SUPPORT` option. Please refer to the protocols/http_server/ws_echo_server example which demonstrates usage of the websocket feature.

**Event Handling**

ESP HTTP server has various events for which a handler can be triggered by the *Event Loop library* when the particular event occurs. The handler has to be registered using `esp_event_handler_register()`. This helps in event handling for ESP HTTP server.

`esp_http_server_event_id_t` has all the events which can happen for ESP HTTP server.

Expected data type for different ESP HTTP server events in event loop:

- `HTTP_SERVER_EVENT_ERROR` : `httpd_err_code_t`
- `HTTP_SERVER_EVENT_START` : `NULL`
- `HTTP_SERVER_EVENT_ON_CONNECTED` : `int`
- `HTTP_SERVER_EVENT_ON_HEADER` : `int`
- `HTTP_SERVER_EVENT_HEADERS_SENT` : `int`
- `HTTP_SERVER_EVENT_ON_DATA` : `esp_http_server_event_data`
- `HTTP_SERVER_EVENT_SENT_DATA` : `esp_http_server_event_data`
- `HTTP_SERVER_EVENT_DISCONNECTED` : `int`
- `HTTP_SERVER_EVENT_STOP` : `NULL`

**API Reference**

**Header File**

- `components/esp_http_server/include/esp_http_server.h`
- This header file can be included with:

```c
#include "esp_http_server.h"
```

- This header file is a part of the API provided by the `esp_http_server` component. To declare that your component depends on `esp_http_server`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_http_server
```

or

```cmake
PRIV_REQUIRES esp_http_server
```

**Functions**

`esp_err_t httpd_register_uri_handler (httpd_handle_t handle, const httpd_uri_t *uri_handler)`

Registers a URI handler.

Example usage:

```c
esp_err_t my_uri_handler (httpd_req_t * req)
{
    // Recv, Process and Send
    ...
    ...
}(continues on next page)```
// Fail condition
if (...) {
    // 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.

Parameters
- **handle** - [in] handle to HTTPD server instance
- **uri_handler** - [in] pointer to handler that needs to be registered

Returns
- 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

```c
esp_err_t httpd_unregister_uri_handler(httpd_handle_t handle, const char *uri, httpd_method_t method)
```

Unregister a URI handler.

Parameters
- **handle** - [in] handle to HTTPD server instance
- **uri** - [in] URI string
- **method** - [in] HTTP method

Returns
- 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

```c
esp_err_t httpd_unregister_uri(httpd_handle_t handle, const char *uri)
```

Unregister all URI handlers with the specified uri string.

Parameters
- **handle** - [in] handle to HTTPD server instance
- **uri** - [in] uri string specifying all handlers that need to be deregistered

Returns
• 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

**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()

**Parameters**

- `hd` - [in] HTTPD instance handle
- `sockfd` - [in] Session socket FD
- `recv_func` - [in] The receive function to be set for this session

**Returns**

- ESP_OK : On successfully registering override
- ESP_ERR_INVALID_ARG : Null arguments

**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()

**Parameters**

- `hd` - [in] HTTPD instance handle
- `sockfd` - [in] Session socket FD
- `send_func` - [in] The send function to be set for this session

**Returns**

- ESP_OK : On successfully registering override
- ESP_ERR_INVALID_ARG : Null arguments

**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()

**Parameters**

- `hd` - [in] HTTPD instance handle
• **sockfd** -[in] Session socket FD
• **pending_func** -[in] The receive function to be set for this session

**Returns**
• ESP_OK : On successfully registering override
• ESP_ERR_INVALID_ARG : Null arguments

```
esp_err_t httpd_req_async_handler_begin(httpd_req_t *r, httpd_req_t **out)
```

Start an asynchronous request. This function can be called in a request handler to get a request copy that can be used on an async thread.

**Note:**
• This function is necessary in order to handle multiple requests simultaneously. See examples/async_requests for example usage.
• You must call `httpd_req_async_handler_complete()` when you are done with the request.

**Parameters**
• **r** -[in] The request to create an async copy of
• **out** -[out] A newly allocated request which can be used on an async thread

**Returns**
• ESP_OK : async request object created

```
esp_err_t httpd_req_async_handler_complete(httpd_req_t *r)
```

Mark an asynchronous request as completed. This will.

• free the request memory
• relinquish ownership of the underlying socket, so it can be reused.
• allow the http server to close our socket if needed (lru_purge_enable)

**Note:** If async requests are not marked completed, eventually the server will no longer accept incoming connections. The server will log a “httpd_accept_conn: error in accept (23)” message if this happens.

**Parameters** **r** -[in] The request to mark async work as completed

**Returns**
• ESP_OK : async request was marked completed

```
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.

**Parameters** **r** -[in] The request whose socket descriptor should be found

**Returns**
• Socket descriptor : The socket descriptor for this request
• -1 : Invalid/NULL request pointer
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

Parameters
  - r – [in] The request being responded to
  - buf – [in] Pointer to a buffer that the data will be read into
  - buf_len – [in] Length of the buffer

Returns
  - 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()

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.

Parameters
  - r – [in] The request being responded to
  - field – [in] The header field to be searched in the request

Returns
  - Length: If field is found in the request URL
  - Zero: Field not found / Invalid request / Null arguments

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
Chapter 2. API Reference

Parameters
• r [in] The request being responded to
• field [in] The field to be searched in the header
• val [out] Pointer to the buffer into which the value will be copied if the field is found
• val_size [in] Size of the user buffer “val”

Returns
• 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

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

Parameters r [in] The request being responded to

Returns
• Length: Query is found in the request URL
• Zero: Query not found / Null arguments / Invalid request

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

Parameters
• r [in] The request being responded to
• buf [out] Pointer to the buffer into which the query string will be copied (if found)
• buf_len [in] Length of output buffer

Returns
• 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
**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&param2=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.

### Parameters
- **qry** [in] Pointer to query string
- **key** [in] The key to be searched in the query string
- **val** [out] Pointer to the buffer into which the value will be copied if the key is found
- **val_size** [in] Size of the user buffer "val"

### Returns
- 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

**esp_err_t httpd_req_get_cookie_val** (httpd_req_t *req, const char*cookie_name, char*val, size_t*val_size)

Get the value string of a cookie value from the “Cookie” request headers by cookie name.

### Parameters
- **req** [in] Pointer to the HTTP request
- **cookie_name** [in] The cookie name to be searched in the request
- **val** [out] Pointer to the buffer into which the value of cookie will be copied if the cookie is found
- **val_size** [inout] Pointer to size of the user buffer “val”. This variable will contain cookie length if ESP_OK is returned and required buffer length incase ESP_ERR_HTTPD_RESULT_TRUNC is returned.

### Returns
- ESP_OK: Key is found in the cookie 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
- ESP_ERR_NO_MEM: Memory allocation failure

**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.

### Parameters
- **uri_template** [in] URI template (pattern)
- **uri_to_match** [in] URI to be matched
Chapter 2. API Reference

• **match_upto** - [in] how many characters of the URI buffer to test (there may be trailing query string etc.)

Returns true if a match was found

```c
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, `httpdRespsetType()` - 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.

**Parameters**

• `r` - [in] The request being responded to
• `buf` - [in] Buffer from where the content is to be fetched
• `buf_len` - [in] Length of the buffer, `HTTPD_RESP_USE_STRLEN` to use `strlen()`

**Returns**

• 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

```c
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, `httpdRespsetType()` - 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.

**Parameters**

• `r` - [in] The request being responded to
Chapter 2. API Reference

- **buf**: [in] Pointer to a buffer that stores the data
- **buf_len**: [in] Length of the buffer, HTTPD_RESP_USE_STRLEN to use strlen()

**Returns**
- 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

static inline esp_err_t httpd_resp_sendstr (httpd_req_t *r, const char *str)

This API simply calls http_resp_send with buffer length set to string length assuming the buffer contains a null terminated string

**Parameters**
- **r**: [in] The request being responded to
- **str**: [in] String to be sent as response body

**Returns**
- 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

static inline esp_err_t httpd_resp_sendstr_chunk (httpd_req_t *r, const char *str)

This API simply calls http_resp_send_chunk with buffer length set to string length assuming the buffer contains a null terminated string

**Parameters**
- **r**: [in] The request being responded to
- **str**: [in] String to be sent as response body (NULL to finish response packet)

**Returns**
- 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

**esp_err_t httpd_resp_set_status (httpd_req_t *r, const char *status)**

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.

**Parameters**
- **r**: [in] The request being responded to
- **status**: [in] The HTTP status code of this response

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Null arguments
• ESP_ERR_HTTPD_INVALID_REQ: Invalid request pointer

```c
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.

**Parameters**
- `r` - [in] The request being responded to
- `type` - [in] The Content Type of the response

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Null arguments
- ESP_ERR_HTTPD_INVALID_REQ: Invalid request pointer

```c
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.

**Parameters**
- `r` - [in] The request being responded to
- `field` - [in] The field name of the HTTP header
- `value` - [in] The value of this HTTP header

**Returns**
- 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

```c
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.

Parameters
  • `req` – [in] Pointer to the HTTP request for which the response needs to be sent
  • `error` – [in] Error type to send
  • `msg` – [in] Error message string (pass NULL for default message)

Returns
  • 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

static inline `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.

Parameters `r` – [in] The request being responded to

Returns
  • 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

static inline `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.
static inline 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.

Parameters

- **r** - [in] The request being responded to

Returns

- 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

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()

Parameters

- **r** - [in] The request being responded to
- **buf** - [in] Buffer from where the fully constructed packet is to be read
- **buf_len** - [in] Length of the buffer

Returns

- 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()

int httpd_socket_send (httpd_handle_t hd, int sockfd, const char *buf, size_t buf_len, int flags)

A low level API to send data on a given socket

This internally calls the default send function, or the function registered by httpd_sess_set_send_override().

Note: This API is not recommended to be used in any request handler. Use this only for advanced use cases, wherein some asynchronous data is to be sent over a socket.
### Parameters
- `hd` - [in] server instance
- `sockfd` - [in] session socket file descriptor
- `buf` - [in] buffer with bytes to send
- `buf_len` - [in] data size
- `flags` - [in] flags for the `send()` function

### Returns
- Bytes: The number of bytes 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()`

```c
int httpd_socket_send(httpd_handle_t hd, int sockfd, char* buf, size_t buf_len, int flags)
```

A low level API to receive data from a given socket

This internally calls the default `recv` function, or the function registered by `httpd_sess_set_recv_override()`.

**Note:** This API is not recommended to be used in any request handler. Use this only for advanced use cases, wherein some asynchronous communication is required.

### Parameters
- `hd` - [in] server instance
- `sockfd` - [in] session socket file descriptor
- `buf` - [in] buffer with bytes to send
- `buf_len` - [in] data size
- `flags` - [in] flags for the `send()` function

### Returns
- Bytes: The number of bytes received 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()`

```c
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.

### Parameters
- `handle` - [in] HTTP server handle
- `error` - [in] Error type
- `handler_fn` - [in] User implemented handler function (Pass NULL to unset any previously set handler)

### Returns
- `ESP_OK`: handler registered successfully
- `ESP_ERR_INVALID_ARG`: invalid error code or server handle

```c
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
    return server;
}

Parameters
• config [in] Configuration for new instance of the server
• handle [out] Handle to newly created instance of the server. NULL on error

Returns
• 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

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);
    }
}

Parameters handle [in] Handle to server returned by httpd_start

Returns
• ESP_OK: Server stopped successfully
• ESP_ERR_INVALID_ARG: Handle argument is Null

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.
Parameters

• `handle` - [in] Handle to server returned by `httpd_start`
• `work` - [in] Pointer to the function to be executed in the HTTPD’s context
• `arg` - [in] Pointer to the arguments that should be passed to this function

Returns

• ESP_OK: On successfully queueing the work
• ESP_FAIL: Failure in ctrl socket
• ESP_ERR_INVALID_ARG: Null arguments

```c
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.

Parameters

• `handle` - [in] Handle to server returned by `httpd_start`

Returns

• `void*`: Pointer to the context associated with this session
• `NULL`: Empty context / Invalid handle / Invalid socket fd

```c
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

• `handle` - [in] Handle to server returned by `httpd_start`
• `sockfd` - [in] The socket descriptor for which the context should be extracted.

Returns

• `void*`: Pointer to the context associated with this session
• `NULL`: Empty context / Invalid handle / Invalid socket fd

```c
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 also:

`httpd_sess_get_ctx()`

Parameters

• `handle` - [in] Handle to server returned by `httpd_start`
• `sockfd` - [in] The socket descriptor for which the context should be extracted.

Returns

• `void*`: Pointer to the transport context associated with this session
• `NULL`: Empty context / Invalid handle / Invalid socket fd

```c
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 also:

`httpd_sess_set_ctx()`

Parameters

• `handle` - [in] Handle to server returned by `httpd_start`
**sockfd**  - [in] The socket descriptor for which the context should be extracted.

**ctx**  - [in] Transport context object to assign to the session

**free_fn**  - [in] Function that should be called to free the transport context

```c
void *httpd_get_global_user_ctx (httpd_handle_t handle)

Get HTTPD global user context (it was set in the server config struct)

Parameters
- handle  - [in] Handle to server returned by httpd_start

Returns
global user context
```

```c
void *httpd_get_global_transport_ctx (httpd_handle_t handle)

Get HTTPD global transport context (it was set in the server config struct)

Parameters
- handle  - [in] Handle to server returned by httpd_start

Returns
global transport context
```

```c
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.
```

```c
Parameters
- handle  - [in] Handle to server returned by httpd_start
- sockfd  - [in] The socket descriptor of the session to be closed

Returns
- 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
```

```c
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 sock-
ets/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.
```

```c
Parameters
- handle  - [in] Handle to server returned by httpd_start
- sockfd  - [in] The socket descriptor of the session for which LRU counter is to be updated

Returns
- ESP_OK : Socket found and LRU counter updated
- ESP_ERR_NOT_FOUND : Socket not found
- ESP_ERR_INVALID_ARG : Null arguments
```

```c
esp_err_t httpd_get_client_list (httpd_handle_t handle, size_t *fds, int *client_fds)

Returns list of current socket descriptors of active sessions.
```
**Note:** Size of provided array has to be equal or greater then maximum number of opened sockets, configured upon initialization with `max_open_sockets` field in `httpd_config_t` structure.

**Parameters**

- **handle** - `[in]` Handle to server returned by `httpd_start`
- **fds** - `[inout]` In: Size of provided `client_fds` array Out: Number of valid client fds returned in `client_fds`,
- **client_fds** - `[out]` Array of client fds

**Returns**

- ESP_OK : Successfully retrieved session list
- ESP_ERR_INVALID_ARG : Wrong arguments or list is longer than provided array

**Structures**

```c
struct esp_http_server_event_data
```

Argument structure for `HTTP_SERVER_EVENT_ON_DATA` and `HTTP_SERVER_EVENT_SENT_DATA` event

**Public Members**

- **int fd**
  - Session socket file descriptor
- **int data_len**
  - Data length

```c
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 (3 sockets are reserved for internal working of the HTTP server)

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

bool enable_so_linger
    bool to enable/disable linger

int linger_timeout
    linger timeout (in seconds)
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

`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.

The server will only close the socket if no custom session closing callback is set. If a custom callback is
used, close(sockfd) should be called in here for most cases.

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

ESP_HTTPD_DEF_CTRL_PORT
    HTTP Server control socket port

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 HTTPD low-level send function.
Note: User specified send function must handle errors internally, depending upon the set value of errno, and return specific HTTPD_SOCK_ERR_ codes, which will eventually be conveyed as return value of httpd_send() function

```c
typedef int(*httpd_send_func_t)(httpd_handle_t hd, int sockfd, char* buf, size_t buf_len, int flags)
```
Prototype for HTTPDs low-level send function.

Note: User specified recv function must handle errors internally, depending upon the set value of errno, and return specific HTTPD_SOCK_ERR_ codes, which will eventually be conveyed as return value of httpd_req_recv() function

```c
Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Param buf [in] buffer with bytes to send
Param buf_len [in] data size
Param flags [in] flags for the send() function
Return
• Bytes : The number of bytes 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()
```

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 pending function must handle errors internally, depending upon the set value of errno, and return specific HTTPD_SOCK_ERR_ codes, which will be handled accordingly in the server task.

```c
Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Param buf [in] buffer with bytes to send
Param buf_len [in] data size
Param flags [in] flags for the send() function
Return
• Bytes : The number of bytes received 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()
```

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_SOCK_ERR_ codes, which will be handled accordingly in the server task.

```c
Param hd [in] server instance
Param sockfd [in] session socket file descriptor
Return
• Bytes : The number of bytes waiting to be received
• HTTPD_SOCK_ERR_INVALID : Invalid arguments
• HTTPD_SOCK_ERR_TIMEOUT : Timeout/interrupted while calling socket pending()
• HTTPD_SOCK_ERR_FAIL : Unrecoverable error while calling socket pending()
```

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, `httpdRespSendErr()` 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.

**Param** `req` [in] HTTP request for which the error needs to be handled

**Param** `error` [in] Error type

**Return**
- `ESP_OK`: error handled successful
- `ESP_FAIL`: failure indicates that the underlying socket needs to be closed

```c
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)

**Param** `ctx` [in] object to free

```c
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/recv functions and other parameters of the socket.

**Param** `hd` [in] server instance
**Param** `sockfd` [in] session socket file descriptor

**Return**
- `ESP_OK`: On success
- Any value other than ESP_OK will signal the server to close the socket immediately

```c
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.

**Param** `hd` [in] server instance
**Param** `sockfd` [in] 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.

Param reference_uri [in] URI/template with respect to which the other URI is matched
Param uri_to_match [in] URI/template being matched to the reference URI/template
Param match_upto [in] 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),

Return true on match

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.

Param arg [in] 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:

enumerator HTTPD_500_INTERNAL_SERVER_ERROR
enumerator HTTPD_501_METHOD_NOT_IMPLEMENTED
enumerator HTTPD_505_VERSION_NOT_SUPPORTED
enumerator HTTPD_400_BAD_REQUEST
enumerator HTTPD_401_UNAUTHORIZED
enumerator HTTPD_403_FORBIDDEN
enumerator HTTPD_404_NOT_FOUND
enumerator HTTPD_405_METHOD_NOT_ALLOWED
enumerator HTTPD_408_REQ_TIMEOUT
enumerator HTTPD_411_LENGTH_REQUIRED
enumerator HTTPD_414_URI_TOO_LONG

enumerator HTTPD_431_REQ_HDR_FIELDS_TOO_LARGE

enumerator HTTPD_ERR_CODE_MAX

enum esp_http_server_event_id_t

HTTP Server events id.

Values:

enumerator HTTP_SERVER_EVENT_ERROR
    This event occurs when there are any errors during execution

enumerator HTTP_SERVER_EVENT_START
    This event occurs when HTTP Server is started

enumerator HTTP_SERVER_EVENT_ON_CONNECTED
    Once the HTTP Server has been connected to the client, no data exchange has been performed

enumerator HTTP_SERVER_EVENT_ON_HEADER
    Occurs when receiving each header sent from the client

enumerator HTTP_SERVER_EVENT_HEADERS_SENT
    After sending all the headers to the client

enumerator HTTP_SERVER_EVENT_ON_DATA
    Occurs when receiving data from the client

enumerator HTTP_SERVER_EVENT_SENT_DATA
    Occurs when an ESP HTTP server session is finished

enumerator HTTP_SERVER_EVENT_DISCONNECTED
    The connection has been disconnected

enumerator HTTP_SERVER_EVENT_STOP
    This event occurs when HTTP Server is stopped

2.2.10 HTTPS Server

Overview

This component is built on top of HTTP Server. The HTTPS server takes advantage of hook registration functions in the regular HTTP server to provide callback function for SSL session.

All documentation for HTTP Server applies also to a server you create this way.
Chapter 2. API Reference

Used APIs

The following APIs of HTTP Server should not be used with HTTPS Server, as they are used internally to handle secure sessions and to maintain internal state:

- “send” , “receive” and “pending” callback registration functions - secure socket handling
  - httpd_sess_set_send_override()
  - httpd_sess_set_recv_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::global_transport_ctx
  - httpd_config::global_transport_ctx_free_fn
  - httpd_config::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 into the firmware, and pass the init struct into the start function after the certificate address and lengths are correctly configured in 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 speed or more verbose logging. Subsequent requests through the open secure socket are much faster (down to under 100 ms).

API Reference

Header File

- components/esp_https_server/include/esp_https_server.h
- This header file can be included with:

```
#include "esp_https_server.h"
```

- This header file is a part of the API provided by the esp_https_server component. To declare that your component depends on esp_https_server, add the following to your CMakeLists.txt:

```
REQUIRES esp_https_server
```

or

```
PRIV_REQUIRES esp_https_server
```

Functions

httpd_ssl_config_t httpd_ssl_config_default (void)

Returns the httpd config struct with default initialisation

**Returns**

- httpd_ssl_config_t HTTPD ssl config struct with default initialisation

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

```c
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)

**Parameters**
- `config` - [inout] - server config, must not be const. Does not have to stay valid after calling this function.
- `handle` - [out] - storage for the server handle, must be a valid pointer

**Returns**
- success

```c
esp_err_t httpd_ssl_stop(httpd_handle_t handle)
```
Stop the server. Blocks until the server is shut down.

**Parameters**
- `handle` - [in]

**Returns**
- ESP_OK: Server stopped successfully
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_FAIL: Failure to shutdown server

**Structures**

```c
struct esp_https_server_user_cb_arg
```
Callback data struct, contains the ESP-TLS connection handle and the connection state at which the callback is executed.

**Public Members**

```c
httpd_ssl_user_cb_state_t user_cb_state
```
State of user callback

```c
esp_tls_t *tls
```
ESP-TLS connection handle

```c
struct httpd_ssl_config
```
HTTPS server config struct

Please use HTTPD_SSL_CONFIG_DEFAULT() to initialize it.

**Public Members**

```c
httpd_config_t httpd
```
Underlying HTTPD server config

Parameters like task stack size and priority can be adjusted here.

```c
const uint8_t *servercert
```
Server certificate

```c
size_t servercert_len
```
Server certificate byte length
const uint8_t *cacert_pem  
CA certificate ((CA used to sign clients, or client cert itself)

size_t cacert_len  
CA certificate byte length

const uint8_t *prvtkey_pem  
Private key

size_t prvtkey_len  
Private key byte length

bool use_ecdsa_peripheral  
Use ECDSA peripheral to use private key

uint8_t ecdsa_key_efuse_blk  
The efuse block where ECDSA key is stored

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)

bool session_tickets  
Enable tls session tickets

bool use_secure_element  
Enable secure element for server session

esp_https_server_user_cb *user_cb  
User callback for esp_https_server

void *ssl_userdata  
user data to add to the ssl context

const char **alpn_protos  
Application protocols the server supports in order of preference. Used for negotiating during the TLS handshake, first one the client supports is selected. The data structure must live as long as the https server itself!

 Macros

HTTPD_SSL_CONFIG_DEFAULT
Type Definitions

typedef struct *esp_https_server_user_cb_arg esp_https_server_user_cb_arg_t

Callback data struct, contains the ESP-TLS connection handle and the connection state at which the callback is executed.

typedef void esp_https_server_user_cb (esp_https_server_user_cb_arg_t *user_cb)

Callback function prototype Can be used to get connection or client information (SSL context) E.g. Client certificate, Socket FD, Connection state, etc.

    Param user_cb Callback data struct

typedef struct httpd_ssl_config httpd_ssl_config_t

Enumerations

enum httpd_ssl_transport_mode_t

    Values:

enumerator HTTPD_SSL_TRANSPORT_SECURE

enumerator HTTPD_SSL_TRANSPORT_INSECURE

enum httpd_ssl_user_cb_state_t

    Indicates the state at which the user callback is executed, i.e. at session creation or session close.

    Values:

enumerator HTTPD_SSL_USER_CB_SESS_CREATE

enumerator HTTPD_SSL_USER_CB_SESS_CLOSE

2.2.11  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 or 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:

```c
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_us;
    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_us, sizeof(total_time_us));
    printf("%d packets transmitted, %d received, time %dms\n", transmitted, received, total_time_us);
}

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)
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 does not 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 will not 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

- `components/lwip/include/apps/ping/ping_sock.h`
  - This header file can be included with:

    ```c
    #include "ping/ping_sock.h"
    ```

  - This header file is a part of the API provided by the `lwip` component. To declare that your component depends on `lwip`, add the following to your CMakeLists.txt:

    ```cmake
    REQUIREES lwip
    ```

    or

    ```cmake
    PRIV_REQUIRES lwip
    ```
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.

**Parameters**
- `config` - ping configuration
- `cbs` - a bunch of callback functions invoked by internal ping task
- `hdl_out` - handle of ping session

**Returns**
- 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

`esp_err_t esp_ping_delete_session(esp_ping_handle_t hdl)`

Delete a ping session.

**Parameters**
- `hdl` - handle of ping session

**Returns**
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: delete ping session successfully

`esp_err_t esp_ping_start(esp_ping_handle_t hdl)`

Start the ping session.

**Parameters**
- `hdl` - handle of ping session

**Returns**
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: start ping session successfully

`esp_err_t esp_ping_stop(espPingHandle_t hdl)`

Stop the ping session.

**Parameters**
- `hdl` - handle of ping session

**Returns**
- ESP_ERR_INVALID_ARG: invalid parameters (e.g. ping handle is null, etc)
- ESP_OK: stop ping session successfully

`esp_err_t esp_ping_get_profile(esp_ping_handle_t hdl, espPingProfile_t profile, void *data, uint32_t size)`

Get runtime profile of ping session.

**Parameters**
- `hdl` - handle of ping session
- `profile` - type of profile
- `data` - profile data
- `size` - profile data size

**Returns**
- 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

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

int tos
    Type of Service, a field specified in the IP header

int ttl
    Time to Live, 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

uint32_t interface
    Netif index, interface=0 means NETIF_NO_INDEX
Chapter 2. API Reference

### 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:**

- **enumerator ESP_PING_PROF_SEQNO**
  Sequence number of a ping procedure
- **enumerator ESP_PING_PROF_TOS**
  Type of service of a ping procedure
- **enumerator ESP_PING_PROF_TTL**
  Time to live of a ping procedure
- **enumerator ESP_PING_PROF_REQUEST**
  Number of request packets sent out
- **enumerator ESP_PING_PROF_REPLY**
  Number of reply packets received
- **enumerator ESP_PING_PROF_IPADDR**
  IP address of replied target
- **enumerator ESP_PING_PROF_SIZE**
  Size of received packet
- **enumerator ESP_PING_PROF_TIMEGAP**
  Elapsed time between request and reply packet
- **enumerator ESP_PING_PROF_DURATION**
  Elapsed time of the whole ping session

### 2.2.12 mDNS Service

mDNS is a multicast UDP service that is used to provide local network service and host discovery. The ESP-IDF component mDNS has been moved from ESP-IDF since version v5.0 to a separate repository:
Chapter 2. API Reference

- mDNS component on GitHub

To add mDNS component in your project, please run `idf.py add-dependency espressif/mdns`.

Hosted Documentation

The documentation can be found on the link below:

- mDNS documentation

2.2.13 Mbed TLS

Mbed TLS is a C library that implements cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols. Its small code footprint makes it suitable for embedded systems.

Note: ESP-IDF uses a fork of Mbed TLS which includes a few patches (related to hardware routines of certain modules like bignum (MPI) and ECC) over vanilla Mbed TLS.

Mbed TLS supports SSL 3.0 up to TLS 1.3 and DTLS 1.0 to 1.2 communication by providing the following:

- TCP/IP communication functions: listen, connect, accept, read/write.
- SSL/TLS communication functions: init, handshake, read/write.
- X.509 functions: CRT, CRL and key handling
- Random number generation
- Hashing
- Encryption/decryption

Supported TLS versions include SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3, but on the latest ESP-IDF, SSL 3.0, TLS 1.0, and TLS 1.1 have been removed from Mbed TLS. Supported DTLS versions include DTLS 1.0, DTLS 1.1, and DTLS 1.2, but on the latest ESP-IDF, DTLS 1.0 has been removed from Mbed TLS.

Mbed TLS Documentation

For Mbed TLS documentation please refer to the following (upstream) pointers:

- API Reference
- Knowledge Base

Mbed TLS Support in ESP-IDF

Please find the information about the Mbed TLS versions presented in different branches of ESP-IDF here.

Note: Please refer the Mbed TLS to migrate from Mbed TLS version 2.x to version 3.0 or greater.

Application Examples

Examples in ESP-IDF use ESP-TLS which provides a simplified API interface for accessing the commonly used TLS functionality.

Refer to the examples protocols/https_server/simple (Simple HTTPS server) and protocols/https_request (Make HTTPS requests) for more information.

If the Mbed TLS API is to be used directly, refer to the example protocols/https_mbedtls.
Alternatives

**ESP-TLS** acts as an abstraction layer over the underlying SSL/TLS library and thus has an option to use Mbed TLS or wolfSSL as the underlying library. By default, only Mbed TLS is available and used in ESP-IDF whereas wolfSSL is available publicly at [https://github.com/espressif/esp-wolfSSL](https://github.com/espressif/esp-wolfSSL) with the upstream submodule pointer.

Please refer to [ESP-TLS: Underlying SSL/TLS Library Options](#) docs for more information on this and comparison of Mbed TLS and wolfSSL.

**Important Config Options**

Following is a brief list of important config options accessible at Component Config -> mbedTLS. The full list of config options can be found [here](#).

- **CONFIG_MBEDTLS_SSL_PROTO_TLS1_2**: Support for TLS 1.2
- **CONFIG_MBEDTLS_SSL_PROTO_TLS1_3**: Support for TLS 1.3
- **CONFIG_MBEDTLS_CERTIFICATE_BUNDLE**: Support for trusted root certificate bundle (more about this: ESP x509 Certificate Bundle)
- **CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS**: Support for TLS Session Resumption: Client session tickets
- **CONFIG_MBEDTLS_SERVER_SSL_SESSION_TICKETS**: Support for TLS Session Resumption: Server session tickets
- **CONFIG_MBEDTLS_HARDWARE_SHA**: Support for hardware SHA acceleration
- **CONFIG_MBEDTLS_HARDWARE_AES**: Support for hardware AES acceleration
- **CONFIG_MBEDTLS_HARDWARE_MPI**: Support for hardware MPI (bignum) acceleration

**Note:** Mbed TLS v3.0.0 and later support only TLS 1.2 and TLS 1.3 (SSL 3.0, TLS 1.0, TLS 1.1, and DTLS 1.0 are not supported). The support for TLS 1.3 is experimental and only supports the client-side. More information about this can be found out [here](#).

**Performance and Memory Tweaks**

**Reducing Heap Usage** The following table shows typical memory usage with different configs when the protocols/https_request example (with Server Validation enabled) was run with Mbed TLS as the SSL/TLS library.

<table>
<thead>
<tr>
<th>Mbed TLS Test</th>
<th>Related Configs</th>
<th>Heap Usage (approx.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>NA</td>
<td>42196 B</td>
</tr>
<tr>
<td>Enable SSL Variable Length</td>
<td><strong>CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH</strong></td>
<td>42120 B</td>
</tr>
<tr>
<td>Disable Keep Peer Certificate</td>
<td><strong>CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE</strong></td>
<td>38533 B</td>
</tr>
<tr>
<td>Enable Dynamic TX/RX Buffer</td>
<td><strong>CONFIG_MBEDTLS_DYNAMIC_BUFFER</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>CONFIG_MBEDTLS_DYNAMIC_FREE_CA_CERT</strong></td>
<td>22013 B</td>
</tr>
</tbody>
</table>

**Note:** These values are subject to change with change in configuration options and versions of Mbed TLS.

**Reducing Binary Size** Under Component Config -> mbedTLS, there are multiple Mbed TLS features which are enabled by default but can be disabled if not needed to save code size. More information can be about this can be found in [Minimizing Binary Size](#) docs.

Code examples for this API section are provided in the protocols directory of ESP-IDF examples.
2.2.14 IP Network Layer

Documentation for IP Network Layer protocols (below the Application Protocol layer) are provided in Networking APIs.

2.3 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** (0x10c): Operation has not fully completed
- **ESP_ERR_NOT_ALLOWED** (0x10d): Operation is not allowed
- **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): A requested entry couldn’t be found or 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): Value doesn’t fit into the entry or 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_NVS_WRONG_ENCRYPTION (0x1119): NVS partition is marked as encrypted with generic flash encryption. This is forbidden since the NVS encryption works differently.

ESP_ERR_ULT_BASE (0x1200): Offset for ULP-related error codes

ESP_ERR_ULT_SIZE_TOO_BIG (0x1201): Program doesn’t fit into RTC memory reserved for the ULP

ESP_ERR_ULT_INVALID_LOAD_ADDR (0x1202): Load address is outside of RTC memory reserved for the ULP

ESP_ERR_ULT_DUPLICATE_LABEL (0x1203): More than one label with the same number was defined

ESP_ERR_ULT_UNDEFINED_LABEL (0x1204): Branch instructions references an undefined label

ESP_ERR_ULT_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_NOT_ENOUGH_UNUSED_KEY_BLOCKS** (0x1605): Error not enough unused key blocks available

**ESP_ERR_DAMAGED_READING** (0x1606): Error. Burn or reset was done during a reading operation leads to damage read data. This error is internal to the efuse component and not returned by any public API.

**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_NV** (0x3008): WiFi internal NVS module error

**ESP_ERR_WIFI_MAC** (0x3009): MAC address is invalid

**ESP_ERR_WIFISSID** (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_NOT_ASSOC** (0x3015): The WiFi connection is not associated

**ESP_ERR_WIFI_TX_DISALLOW** (0x3016): The WiFi TX is disallowed

**ESP_ERR_WIFI_TWT_FULL** (0x3017): no available flow id

**ESP_ERR_WIFI_TWT_SETUP_TIMEOUT** (0x3018): Timeout of receiving twt setup response frame, timeout times can be set during twt setup

**ESP_ERR_WIFI_TWT_SETUP_TXFAIL** (0x3019): TWT setup frame tx failed

**ESP_ERR_WIFI_TWT_SETUP_REJECT** (0x301a): The twt setup request was rejected by the AP

**ESP_ERR_WIFI_DISCARD** (0x301b): Discard frame

**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_DPP_FAILURE (0x3097): Generic failure during DPP Operation
ESP_ERR_DPP_TX_FAILURE (0x3098): DPP Frame Tx failed or not Acked
ESP_ERR_DPP_INVALID_ATTR (0x3099): Encountered invalid DPP Attribute
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_PS (0x4019)
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_DHCPC_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_ESP_NETIF_MLD6_FAILED (0x500b)
ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED (0x500c)
ESP_ERR_ESP_NETIF_DHCP5S_START_FAILED (0x500d)
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_INITIATED (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_HTTP_CONNECTION_CLOSED (0x7008): Read FIN from peer and the connection closed
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/get socket option
ESP_ERR_ESP_TLS_CONNECTION_TIMEOUT (0x8006): new connection in esp_tls_low_level_conn connection timedout
ESP_ERR_ESP_TLS_SE_FAILED (0x8007)
Chapter 2. API Reference

ESP_ERR_ESP_TLS_TCP_CLOSED_FIN (0x8008)
ESP_ERR_MBEDTLS_CERT_PARTLY_OK (0x8010): mbedtls parse certificates was partly successful
ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED (0x8011): mbedtls api returned error
ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED (0x8012): mbedtls api returned error
ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED (0x8013): mbedtls api returned error
ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOLS_FAILED (0x8014): mbedtls api returned error
ESP_ERR_MBEDTLS_X509_CERT_PARSE_FAILED (0x8015): mbedtls api returned error
ESP_ERR_MBEDTLS_SSL_CONF_OWNER_CERT_FAILED (0x8016): mbedtls api returned error
ESP_ERR_MBEDTLS_SSL_SETUP_FAILED (0x8017): mbedtls api returned error
ESP_ERR_MBEDTLS_SSL_WRITE_FAILED (0x8018): mbedtls api returned error
ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED (0x8019): mbedtls api returned failed
ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED (0x801a): mbedtls api returned failed
ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED (0x801b): mbedtls api returned failed
ESP_ERR_MBEDTLS_SSL_TICKET_SETUP_FAILED (0x801c): mbedtls api returned failed
ESP_ERR_WOLFSSL_SSL_SET_HOSTNAME_FAILED (0x8031): wolfSSL api returned error
ESP_ERR_WOLFSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED (0x8032): wolfSSL api returned error
ESP_ERR_WOLFSSL_CERT_VERIFY_SETUP_FAILED (0x8033): wolfSSL api returned error
ESP_ERR_WOLFSSL_KEY_VERIFY_SETUP_FAILED (0x8034): wolfSSL api returned error
ESP_ERR_WOLFSSL_SSL_HANDSHAKE_FAILED (0x8035): wolfSSL api returned failed
ESP_ERR_WOLFSSL_CTX_SETUP_FAILED (0x8036): wolfSSL api returned failed
ESP_ERR_WOLFSSL_SSL_SETUP_FAILED (0x8037): wolfSSL api returned failed
ESP_ERR_WOLFSSL_SSL_WRITE_FAILED (0x8038): wolfSSL api returned failed
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
ESP_ERR_HW_CRYPTO_BASE (0xc000): Starting number of HW cryptography module error codes
ESP_ERR_HW_CRYPTO_DS_HMAC_FAIL (0xc001): HMAC peripheral problem
2.4 Networking APIs

2.4.1 Wi-Fi

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:

```
| MAC Header | Category Code | Organization Identifier | Random Values | Vendor Specific Content | FCS |
```

- 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 | Length | Organization Identifier | Type | Version | Body |
```

- 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:0xff).

Security  
ESP-NOW uses the CCMP method, which is described in IEEE Std. 802.11-2012, to protect the vendor-specific action frame. The Wi-Fi device maintains a Primary Master Key (PMK) and several Local Master Keys (LMK). The lengths of both PMK and LMK are 16 bytes.

- 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 Deinitialization  
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 are 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. 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.

The maximum number of paired devices is 20, and the paired encryption devices are no more than 17, the default is 7. If you want to change the number of paired encryption devices, set `CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM` in the Wi-Fi component configuration menu.
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 does not 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.

### Config ESP-NOW Rate
Call `esp_wifi_config_espnow_rate()` to config ESP-NOW rate of specified interface. Make sure that the interface is enabled before config rate. This API should be called after `esp_wifi_start()`.

### Config ESP-NOW Power-saving Parameter
Sleep is supported only when ESP32-S2 is configured as station. Call `esp_now_setWakeWindow()` to configure Window for ESP-NOW RX at sleep. The default value is the maximum, which allowing RX all the time.

If Power-saving is needed for ESP-NOW, call `esp_wifi_connectionless_module_setWakeInterval()` to configure Interval as well.

Please refer to [connectionless module power save](#) to get more detail.

### Application Examples
- Example of sending and receiving ESP-NOW data between two devices: wifi/espnow.
- For more application examples of how to use ESP-NOW, please visit ESP-NOW repository.

### API Reference

#### Header File
- `components/esp_wifi/include/esp_now.h`
- This header file can be included with:

  ```
  #include "esp_now.h"
  ```

- This header file is a part of the API provided by the `esp_wifi` component. To declare that your component depends on `esp_wifi`, add the following to your CMakeLists.txt:

  ```
  ```
Chapter 2. API Reference

REQUIRES esp_wifi

or

PRIV_REQUIRES esp_wifi

**Functions**

```c
esp_err_t esp_now_init (void)
```

Initialize ESPNOW function.

**Returns**

- ESP_OK : succeed
- ESP_ERR_ESPNOW_INTERNAL : Internal error

```c
esp_err_t esp_now_deinit (void)
```

De-initialize ESPNOW function.

**Returns**

- ESP_OK : succeed

```c
esp_err_t esp_now_get_version (uint32_t *version)
```

Get the version of ESPNOW.

**Parameters**

- `version` – ESPNOW version

**Returns**

- ESP_OK : succeed
- ESP_ERR_ESPNOW_ARG : invalid argument

```c
esp_err_t esp_now_register_recv_cb (esp_now_recv_cb_t cb)
```

Register callback function of receiving ESPNOW data.

**Parameters**

- `cb` – callback function of receiving ESPNOW data

**Returns**

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_INTERNAL : internal error

```c
esp_err_t esp_now_unregister_recv_cb (void)
```

Unregister callback function of receiving ESPNOW data.

**Returns**

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

```c
esp_err_t esp_now_register_send_cb (esp_now_send_cb_t cb)
```

Register callback function of sending ESPNOW data.

**Parameters**

- `cb` – callback function of sending ESPNOW data

**Returns**

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized
- ESP_ERR_ESPNOW_INTERNAL : internal error

```c
esp_err_t esp_now_unregister_send_cb (void)
```

Unregister callback function of sending ESPNOW data.

**Returns**

- ESP_OK : succeed
- ESP_ERR_ESPNOW_NOT_INIT : ESPNOW is not initialized

```c
esp_err_t esp_now_send (const uint8_t *peer_addr, const uint8_t *data, size_t len)
```

Send ESPNOW data.
Chapter 2. API Reference

**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

**Parameters**
- **peer_addr** – peer MAC address
- **data** – data to send
- **len** – length of data

**Returns**
- 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, when this happens, you can delay a while before sending the next data
- ESP_ERR_ESPNOW_NOT_FOUND: peer is not found
- ESP_ERR_ESPNOW_IF: current WiFi interface doesn’t match that of peer

```c
esp_err_t esp_now_add_peer (const esp_now_peer_info_t *peer)
```
Add a peer to peer list.

**Parameters** peer – peer information

**Returns**
- 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

```c
esp_err_t esp_now_del_peer (const uint8_t* peer_addr)
```
Delete a peer from peer list.

**Parameters** peer_addr – peer MAC address

**Returns**
- 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

```c
esp_err_t esp_now_mod_peer (const esp_now_peer_info_t *peer)
```
Modify a peer.

**Parameters** peer – peer information

**Returns**
- 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

```c
esp_err_t esp_wifi_config_espnow_rate (wifi_interface_t ifx, wifi_phy_rate_t rate)
```
Config ESPNOW rate of specified interface.

*Deprecated:* please use `esp_now_set_peer_rate_config()` instead.

**Attention** 1. This API should be called after `esp_wifi_start()`.
**Attention** 2. This API only work when not use Wi-Fi 6 and `esp_now_set_peer_rate_config()` not called.

**Parameters**
- `ifx` - Interface to be configured.
- `rate` - Phy rate to be configured.

**Returns**
- ESP_OK: succeed
- others: failed

```c
esp_err_t esp_now_set_peer_rate_config (const uint8_t*peer_addr, esp_now_rate_config_t *config)
```
Set ESPNOW rate config for each peer.

**Attention** 1. This API should be called after `esp_wifi_start()` and `esp_now_init()`.

**Parameters**
- `peer_addr` - peer MAC address
- `config` - rate config to be configured.

**Returns**
- ESP_OK: succeed
- ESP_ERR_ESPNOW_NOT_INIT: ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG: invalid argument
- ESP_ERR_ESPNOW_INTERNAL: internal error

```c
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.

**Parameters**
- `peer_addr` - peer MAC address
- `peer` - peer information

**Returns**
- 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

```c
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.

**Parameters**
- `from_head` - fetch from head of list or not
- `peer` - peer information

**Returns**
- 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

```c
bool esp_now_is_peer_exist (const uint8_t*peer_addr)
```
Peer exists or not.

**Parameters** `peer_addr` - peer MAC address

**Returns**
- true : peer exists
- false : peer not exists

```c
esp_err_t esp_now_get_peer_num (esp_now_peer_num_t *num)
```
Get the number of peers.
Chapter 2. API Reference

Parameters

**num** - number of peers

Returns

- ESP_OK: succeed
- ESP_ERR_ESPNOW_NOT_INIT: ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG: invalid argument

```c
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

Parameters **pmk** - primary master key

Returns

- ESP_OK: succeed
- ESP_ERR_ESPNOW_NOT_INIT: ESPNOW is not initialized
- ESP_ERR_ESPNOW_ARG: invalid argument

```c
esp_err_t esp_now_set_wake_window(uint16_t window)
```

Set wake window for esp_now to wake up in interval unit.

**Attention** 1. This configuration could work at connected status. When ESP_WIFI_STA_DISCONNECTED_PM_ENABLE is enabled, this configuration could work at disconnected status.  
**Attention** 2. Default value is the maximum.

Parameters **window** – Milliseconds would the chip keep waked each interval, from 0 to 65535.

Returns

- ESP_OK: succeed
- ESP_ERR_ESPNOW_NOT_INIT: ESPNOW is not initialized

**Structures**

```c
struct esp_now_peer_info
```

ESPNOW peer information parameters.

**Public Members**

```c
uint8_t peer_addr[ESP_NOW_ETH_ALEN]
```

ESPNOW peer MAC address that is also the MAC address of station or softap

```c
uint8_t lmk[ESP_NOW_KEY_LEN]
```

ESPNOW peer local master key that is used to encrypt data

```c
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.

```c
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

struct esp_now_recv_info
    ESPNOW packet information.

Public Members

uint8_t *src_addr
    Source address of ESPNOW packet

uint8_t *des_addr
    Destination address of ESPNOW packet

wifi_pkt_rx_ctrl_t *rx_ctrl
    Rx control info of ESPNOW packet

struct esp_now_rate_config
    ESPNOW rate config.

Public Members

wifi_phy_mode_t phymode
    ESPNOW phymode of specified interface

wifi_phy_rate_t rate
    ESPNOW rate of specified interface

bool ersu
    ESPNOW using ersu send frame

bool dcm
    ESPNOW using dcm rate to send frame
Chapter 2. API Reference

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 struct esp_now_recv_info esp_now_recv_info_t
    ESPNOW packet information.

typedef struct esp_now_rate_config esp_now_rate_config_t
    ESPNOW rate config.

typedef void (esp_now_recv_cb_t)(const esp_now_recv_info_t *esp_now_info, const uint8_t *data, int data_len)
    Callback function of receiving ESPNOW data.

    Attention esp_now_info is a local variable, it can only be used in the callback.
    Param esp_now_info received ESPNOW packet information
    Param data received data
    Param 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.
    Param mac_addr peer MAC address
    Param status status of sending ESPNOW data (succeed or fail)

Enumerations

eenum esp_now_send_status_t
    Status of sending ESPNOW data.
    Values:
    enumerator ESP_NOW_SEND_SUCCESS
        Send ESPNOW data successfully
    enumerator ESP_NOW_SEND_FAIL
        Send ESPNOW data fail

ESP-WIFI-MESH Programming Guide

This is a programming guide for ESP-WIFI-MESH, including the API reference and coding examples. This guide is split into the following parts:

1. ESP-WIFI-MESH Programming Model
2. Writing an ESP-WIFI-MESH Application
3. Self-Organized Networking
4. Application Examples
5. API Reference

For documentation regarding the ESP-WIFI-MESH protocol, please see the ESP-WIFI-MESH API Guide. For more information about ESP-WIFI-MESH Development Framework, please see ESP-WIFI-MESH Development Framework.

ESP-WIFI-MESH Programming Model
**Software Stack**  The ESP-WIFI-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-WIFI-MESH software stack.

![Fig. 1: ESP-WIFI-MESH Software Stack](image)

**System Events**  An application interfaces with ESP-WIFI-MESH via ESP-WIFI-MESH Events. Since ESP-WIFI-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-WIFI-MESH application.

![Fig. 2: ESP-WIFI-MESH System Events Delivery](image)

The `mesh_event_id_t` defines all possible ESP-WIFI-MESH events and can indicate events such as the connection/disconnection of parent/child. Before ESP-WIFI-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-WIFI-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.

**Warning:**  When using ESP-WIFI-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-WIFI-MESH’s self-organizing behavior. Therefore, users should not call Wi-Fi APIs after `esp_mesh_start()` is called, and before `esp_mesh_stop()` is called.
**LwIP & ESP-WIFI-MESH**  The application can access the ESP-WIFI-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 that could become root is required to initialize LwIP by calling `esp_netif_init()`. In order to prevent non-root node access to LwIP, the application should not create or register any network interfaces using `esp_netif` APIs.

ESP-WIFI-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-WIFI-MESH Application**  The prerequisites for starting ESP-WIFI-MESH is to initialize LwIP and Wi-Fi. The following code snippet demonstrates the necessary prerequisite steps before ESP-WIFI-MESH itself can be initialized.

```c
ESP_ERROR_CHECK(esp_netif_init());
/* 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-WIFI-MESH network up and running can be summarized into the following three steps:

1. **Initialize Mesh**
2. **Configuring an ESP-WIFI-MESH Network**
3. **Start Mesh**

**Initialize Mesh**  The following code snippet demonstrates how to initialize ESP-WIFI-MESH

```c
/* 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-WIFI-MESH Network**  ESP-WIFI-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-WIFI-MESH:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>Range from 1 to 14</td>
</tr>
<tr>
<td>Mesh ID</td>
<td>ID of ESP-WIFI-MESH Network, see <code>mesh_addr_t</code></td>
</tr>
<tr>
<td>Router</td>
<td>Router Configuration, see <code>mesh_router_t</code></td>
</tr>
<tr>
<td>Mesh AP</td>
<td>Mesh AP Configuration, see <code>mesh_ap_cfg_t</code></td>
</tr>
<tr>
<td>Crypto Functions</td>
<td>Crypto Functions for Mesh IE, see <code>mesh_crypto_funcs_t</code></td>
</tr>
</tbody>
</table>

The following code snippet demonstrates how to configure ESP-WIFI-MESH.
Chapter 2. API Reference

```c
/* 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-WIFI-MESH.

```c
/* mesh start */
ESP_ERROR_CHECK(esp_mesh_start());
```

After starting ESP-WIFI-MESH, the application should check for ESP-WIFI-MESH events to determine when it has connected to the network. After connecting, the application can start transmitting and receiving packets over the ESP-WIFI-MESH network using `esp_mesh_send()` and `esp_mesh_recv()`.

**Self-Organized Networking** Self-organized networking is a feature of ESP-WIFI-MESH where nodes can autonomously scan/select/connect/reconnect to other nodes and routers. This feature allows an ESP-WIFI-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-WIFI-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-WIFI-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-WIFI-MESH.

**Toggling 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 on 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.

```c
//Disable self-organized networking
esp_mesh_set_self_organized(false, false);
```

ESP-WIFI-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-WIFI-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.

<table>
<thead>
<tr>
<th>Select Parent</th>
<th>Is Root Node</th>
<th>Effects</th>
</tr>
</thead>
</table>
| N             | N            | • Nodes already connected to a parent node will remain connected.  
               |              | • Nodes previously scanning for a parent nodes will stop scanning. Call `esp_mesh_connect()` to restart. |
| Y             |              | • A root node already connected to router will stay connected.  
               |              | • A root node disconnected from router will need to call `esp_mesh_connect()` to reconnect. |
| Y             | N            | • 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             |              | • 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.

```c
//Enable self-organized networking and select a new parent
esp_mesh_set_self_organized(true, true);
...
//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-WIFI-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-WIFI-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-WIFI-MESH.

```c
//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); //Do not select new parent
esp_mesh_connect(); //Manually reconnect to router
```

Application Examples  ESP-IDF contains these ESP-WIFI-MESH example projects:

The Internal Communication Example  demonstrates how to set up a ESP-WIFI-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-WIFI-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

- components/esp_wifi/include/esp_mesh.h
- This header file can be included with:

  ```
  #include "esp_mesh.h"
  ```

- This header file is a part of the API provided by the esp_wifi component. To declare that your component depends on esp_wifi, add the following to your CMakeLists.txt:

  ```
  REQUIRES esp_wifi
  ```

  or

Espressif Systems  233  Release master

Submit Document Feedback
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.

**Returns**

- ESP_OK
- ESP_FAIL

**esp_err_t esp_mesh_deinit (void)**

Mesh de-initialization.

- Release resources and stop the mesh

**Returns**

- 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.

**Returns**

- 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.
Chapter 2. API Reference

- Delete TX and RX queues.
- Release resources.
- Restore Wi-Fi softAP to default settings if Wi-Fi dual mode is enabled.
- Set Wi-Fi Power Save type to WIFI_PS_NONE.

**Returns**

- ESP_OK
- ESP_FAIL

```c
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.

**Parameters**

- **to** - [in] 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.

- **data** - [in] 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).

- **flag** - [in] 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_recv_toDS(). Generally, we suggest esp_mesh_recv_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_recv_toDS() which avoids unnecessary memory copy.

- **opt** - [in] 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.
Chapter 2. API Reference

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.

- **opt_count** - [in] option count
  - Currently, this API only takes one option, so opt_count is only supported to be 1.

**Returns**
- 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

```c
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.

**Parameters**
- `time_ms` - [in] blocking time of `esp_mesh_send()`, unit: ms

**Returns**
- ESP_OK

```c
esp_err_t esp_mesh_recv (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.

**Parameters**
- `from` - [out] the address of the original source of the packet
- `data` - [out] 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.
- `timeout_ms` - [in] wait time if a packet isn’t immediately available (0:no wait, port-MAX_DELAY:wait forever)
- `flag` - [out] bitmap for data received
  - MESH_DATA_FROMDS represents data from external IP network
  - MESH_DATA_TODS represents data directed upward within the mesh network
- `opt` - [out] options desired to receive
  - MESH_OPT_RECV_DS_ADDR attaches the DS address
- `opt_count` - [in] option count desired to receive
  - Currently, this API only takes one option, so opt_count is only supported to be 1.

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_START
• ESP_ERR_MESH_TIMEOUT
• ESP_ERR_MESH_DISCARD

`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.

**Parameters**

• `from` [out] the address of the original source of the packet
• `to` [out] the address contains remote IP address and port (IPv4:PORT)
• `data` [out] pointer to the received packet
  – Contain the protocol and applications should follow it to parse the data.
• `timeout_ms` [in] wait time if a packet isn’t immediately available (0:no wait, port-MAX_DELAY:wait forever)
• `flag` [out] 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.
• `opt` [out] options desired to receive
• `opt_count` [in] option count desired to receive

**Returns**

• ESP_OK
• ESP_ERR_MESH_ARGUMENT
• ESP_ERR_MESH_NOT_START
• ESP_ERR_MESH_TIMEOUT
• ESP_ERR_MESH_DISCARD
• ESP_ERR_MESH_RECV_RELEASE

`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.

**Parameters**
- config - [in] pointer to mesh stack configuration

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

```
esp_err_t esp_mesh_get_config(mesh_cfg_t *config)
```

Get mesh stack configuration.

**Parameters**
- config - [out] pointer to mesh stack configuration

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

```
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.

**Parameters**
- router - [in] pointer to router configuration

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

```
esp_err_t esp_mesh_get_router(mesh_router_t *router)
```

Get router configuration.

**Parameters**
- router - [out] pointer to router configuration

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT

```
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.

**Parameters**
- id - [in] pointer to mesh network ID

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT: invalid argument

```
esp_err_t esp_mesh_get_id(mesh_addr_t *id)
```

Get mesh network ID.

**Parameters**
- id - [out] pointer to mesh network ID

**Returns**
Chapter 2. API Reference

- ESP_OK
- ESP_ERR_MESH_ARGUMENT

```c
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

**Parameters**
- `type` - [in] device type

**Returns**
- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED

```c
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.

**Returns** mesh type

```c
esp_err_t esp_mesh_set_max_layer(int max_layer)
```

Set network max layer value.

- for tree topology, the max is 25.
- for chain topology, the max is 1000.
- Network max layer limits the max hop count.

**Attention** This API shall be called before mesh is started.

**Parameters**
- `max_layer` - [in] max layer value

**Returns**
- ESP_OK
- ESP_ERR_MESH_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

```c
int esp_mesh_get_max_layer(void)
```

Get max layer value.

**Returns** max layer value

```c
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.

**Parameters**
- `pwd` - [in] pointer to the password
- `len` - [in] password length

**Returns**
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.

Parameters authmode –[in] authentication mode

Returns
• ESP_OK
• ESP_ERR_MESH_ARGUMENT
• ESP_ERR_MESH_NOT_ALLOWED

wifi_auth_mode_t esp_mesh_get_ap_authmode (void)
Get mesh softAP authentication mode.

Returns authentication mode

esp_err_t esp_mesh_set_ap_connections (int connections)
Set mesh max connection value.

• Set mesh softAP max connection = mesh max connection + non-mesh max connection

Attention  This API shall be called before mesh is started.

Parameters connections –[in] the number of max connections

Returns
• ESP_OK
• ESP_ERR_MESH_ARGUMENT

int esp_mesh_get_ap_connections (void)
Get mesh max connection configuration.

Returns the number of mesh max connections

int esp_mesh_get_non_mesh_connections (void)
Get non-mesh max connection configuration.

Returns the number of non-mesh 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.

Returns 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.
Chapter 2. API Reference

Parameters bssid [out] pointer to parent BSSID

Returns

• ESP_OK
• ESP_FAIL

bool esp_mesh_is_root (void)

Return whether the device is the root node of the network.

Returns 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.

Parameters

• enable [in] enable or disable self-organized networking
• select_parent [in] 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.

Returns

• ESP_OK
• ESP_FAIL

bool esp_mesh_get_self_organized (void)

Return whether enable self-organized networking or not.

Returns 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.

Parameters

• vote [in] 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.
• reason [in] only accept MESH_VOTE_REASON_ROOT_INITIATED for now

Returns
• ESP_OK
• ESP_ERR_MESH_QUEUE_FULL
• ESP_ERR_MESH_DISCARD
• ESP_FAIL

`esp_err_t esp_mesh_set_vote_percentage` (float percentage)
Set vote percentage threshold for approval of being a root (default: 0.9)

• 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.

Parameters percentage [in] vote percentage threshold

Returns
• ESP_OK
• ESP_FAIL

float `esp_mesh_get_vote_percentage` (void)
Get vote percentage threshold for approval of being a root.

Returns 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.

Parameters seconds [in] the expired time

Returns
• ESP_OK
• ESP_FAIL

int `esp_mesh_get_ap_assoc_expire` (void)
Get mesh softAP associate expired time.

Returns 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.

Returns 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)

Returns 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)

Parameters
  • mac -[out] pointer to routing table
  • len -[in] routing table size(in bytes)
  • size -[out] pointer to the number of devices in routing table (including itself)

Returns
  • ESP_OK
  • ESP_ERR_MESH_ARGUMENT

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.

Parameters reachable -[in] this state represents whether the root is able to access external IP network

Returns
  • ESP_OK
  • ESP_FAIL

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.

Parameters pending -[out] pointer to the TX pending

Returns
  • ESP_OK
  • ESP_FAIL

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.

Parameters pending -[out] pointer to the RX pending

Returns
  • ESP_OK
  • ESP_FAIL

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.

Parameters
  • addr -[in] self address or an associate children address
  • xseqno_in -[out] sequence number of the last received packet from the specified address

Returns the number of upQ for a certain address

esp_err_t esp_mesh_set_xon_qsize (int qsize)

Set the number of RX queue for the node, the average number of window allocated to one of its child node is:
wnd = xon_qsize / (2 * max_connection + 1). However, the window of each child node is not strictly equal to
the average value, it is affected by the traffic also.

Attention This API shall be called before mesh is started.

Parameters qsize -[in] default:32 (min:16)

Returns
  • ESP_OK
  • ESP_FAIL
Chapter 2. API Reference

int esp_mesh_get_xon_qsize (void)
  Get queue size.

  Returns  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.

  Parameters allowed [in] allow or not
  Returns
    • ESP_OK
    • ESP_WIFI_ERR_NOT_INIT
    • ESP_WIFI_ERR_NOT_START

bool esp_mesh_is_root_conflicts_allowed (void)
  Check whether allow more than one root to exist in one network.

  Returns  true/false

esp_err_t esp_mesh_set_group_id (const mesh_addr_t *addr, int num)
  Set group ID addresses.

  Parameters
    • addr [in] pointer to new group ID addresses
    • num [in] the number of group ID addresses
  Returns
    • ESP_OK
    • ESP_MESH_ERR_ARGUMENT

esp_err_t esp_mesh_delete_group_id (const mesh_addr_t *addr, int num)
  Delete group ID addresses.

  Parameters
    • addr [in] pointer to deleted group ID address
    • num [in] the number of group ID addresses
  Returns
    • ESP_OK
    • ESP_MESH_ERR_ARGUMENT

int esp_mesh_get_group_num (void)
  Get the number of group ID addresses.

  Returns  the number of group ID addresses

esp_err_t esp_mesh_get_group_list (mesh_addr_t *addr, int num)
  Get group ID addresses.

  Parameters
    • addr [out] pointer to group ID addresses
    • num [in] the number of group ID addresses
  Returns
    • ESP_OK
    • ESP_MESH_ERR_ARGUMENT

bool esp_mesh_is_my_group (const mesh_addr_t *addr)
  Check whether the specified group address is my group.

  Returns  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.
**Parameters**  
num – [in] mesh network capacity

**Returns**
- ESP_OK
- ESP_ERR_MESH_NOT_ALLOWED
- ESP_MESH_ERR_ARGUMENT

```c
int esp_mesh_get_capacity_num(void)
```
Get mesh network capacity.

**Returns**  
mesh network capacity

```c
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 configured.

**Parameters**  
crypto_funcs – [in] crypto functions for mesh IE
- If crypto_funcs is set to NULL, mesh IE is no longer encrypted.

**Returns**
- ESP_OK

```c
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 configured.

**Parameters**
- key – [in] ASCII crypto key
- len – [in] length in bytes, range: 8~64

**Returns**
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

```c
esp_err_t esp_mesh_get_ie_crypto_key(char *key, int len)
```
Get mesh IE crypto key.

**Parameters**
- key – [out] ASCII crypto key
- len – [in] length in bytes, range: 8~64

**Returns**
- ESP_OK
- ESP_MESH_ERR_ARGUMENT

```c
esp_err_t esp_mesh_set_root_healing_delay(int delay_ms)
```
Set delay time before starting root healing.

**Parameters**  
delay_ms – [in] delay time in milliseconds

**Returns**
- ESP_OK

```c
int esp_mesh_get_root_healing_delay(void)
```
Get delay time before network starts root healing.

**Returns**  
delay time in milliseconds

```c
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.

**Parameters** `enable` - [in] enable or not

**Returns**
- ESP_OK

`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.

**Returns** 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.

**Parameters**
- `parent` - [in] 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.
- `parent_mesh_id` - [in] parent mesh ID.
  - If this value is not set, the original mesh ID is used.
- `my_type` - [in] mesh type
  - MESH_STA is not supported.
  - 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.
- `my_layer` - [in] 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.

**Returns**
- ESP_OK
- ESP_ERR_ARGUMENT
- ESP_ERR_MESH_NOT_CONFIG

`esp_err_t esp_mesh_scan_get_ap_ie_len(int *len)`

Get mesh networking IE length of one AP.

**Parameters** `len` - [out] mesh networking IE length

**Returns**
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_INVALID_ARG
- ESP_ERR_WIFI_FAIL

`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.

### Parameters
- `ap_record` [out] pointer to one AP record
- `buffer` [out] pointer to the mesh networking IE of this AP

### Returns
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_INVALID_ARG
- ESP_ERR_WIFI_FAIL

```c
esp_err_t esp_mesh_flush_upstream_packets(void)
```
Flush upstream packets pending in to_parent queue and to_parent_p2p queue.

**Returns**
- ESP_OK

```c
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.

**Parameters**
- `child_mac` [in] an associated child address of this device
- `nodes_num` [out] pointer to the number of nodes in the subnet of a specific child

**Returns**
- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

```c
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.

**Parameters**
- `child_mac` [in] an associated child address of this device
- `nodes` [out] pointer to nodes in the subnet of a specific child
- `nodes_num` [in] the number of nodes in the subnet of a specific child

**Returns**
- ESP_OK
- ESP_ERR_MESH_NOT_START
- ESP_ERR_MESH_ARGUMENT

```c
esp_err_t esp_mesh_disconnect(void)
```
Disconnect from current parent.

**Returns**
- ESP_OK

```c
esp_err_t esp_mesh_connect(void)
```
Connect to current parent.

**Returns**
- ESP_OK

```c
esp_err_t esp_mesh_flush_scan_result(void)
```
Flush scan result.

**Returns**
- 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.

**Parameters**
- `new_bssid` – [in] the new router BSSID if the router changes
- `csa_newchan` – [in] the new channel number to which the whole network is moving
- `csa_count` – [in] channel switch period (beacon count), unit is based on beacon interval of its softAP, the default value is 15.

**Returns**
- ESP_OK

```
esp_err_t esp_mesh_get_router_bssid(uint8_t* router_bssid)
```

Get the router BSSID.

**Parameters** `router_bssid` – [out] pointer to the router BSSID

**Returns**
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_INVALID_ARG

```
int64_t esp_mesh_get_tsf_time(void)
```

Get the TSF time.

**Returns**
- the TSF time

```
esp_err_t esp_mesh_set_topology(esp_mesh_topology_t topo)
```

Set mesh topology. The default value is MESH_TOPO_TREE.

- MESH_TOPO_CHAIN supports up to 1000 layers

**Attention** This API shall be called before mesh is started.

**Parameters** `topo` – [in] MESH_TOPO_TREE or MESH_TOPO_CHAIN

**Returns**
- ESP_OK
- ESP_MESH_ERR_ARGUMENT
- ESP_ERR_MESH_NOT_ALLOWED

```
esp_mesh_topology_t esp_mesh_get_topology(void)
```

Get mesh topology.

**Returns**
- MESH_TOPO_TREE or MESH_TOPO_CHAIN

```
esp_err_t esp_mesh_enable_ps(void)
```

Enable mesh Power Save function.

**Attention** This API shall be called before mesh is started.

**Returns**
### Chapter 2. API Reference

- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_MESH_NOT_ALLOWED

**esp_err_t esp_mesh_disable_ps (void)**

Disable mesh Power Save function.

**Attention** This API shall be called before mesh is started.

**Returns**
- ESP_OK
- ESP_ERR_WIFI_NOT_INIT
- ESP_ERR_MESH_NOT_ALLOWED

**bool esp_mesh_is_ps_enabled (void)**

Check whether the mesh Power Save function is enabled.

**Returns** true/false

**bool esp_mesh_is_device_active (void)**

Check whether the device is in active state.

- If the device is not in active state, it will neither transmit nor receive frames.

**Returns** true/false

**esp_err_t esp_mesh_set_active_duty_cycle (int dev_duty, int dev_duty_type)**

Set the device duty cycle and type.

- The range of dev_duty values is 1 to 100. The default value is 10.
- dev_duty = 100, the PS will be stopped.
- dev_duty is better to not less than 5.
- dev_duty_type could be MESH_PS_DEVICE_DUTY_REQUEST or MESH_PS_DEVICE_DUTY_DEMAND.
- If dev_duty_type is set to MESH_PS_DEVICE_DUTY_REQUEST, the device will use a nwk_duty provided by the network.
- If dev_duty_type is set to MESH_PS_DEVICE_DUTY_DEMAND, the device will use the specified dev_duty.

**Attention** This API can be called at any time after mesh is started.

**Parameters**
- dev_duty - [in] device duty cycle
- dev_duty_type - [in] device PS duty cycle type, not accept MESH_PS_NETWORK_DUTY_MASTER

**Returns**
- ESP_OK
- ESP_FAIL

**esp_err_t esp_mesh_get_active_duty_cycle (int *dev_duty, int *dev_duty_type)**

Get device duty cycle and type.

**Parameters**
- dev_duty - [out] device duty cycle
Chapter 2. API Reference

### dev_duty_type

- **[out]** device PS duty cycle type

**Returns**
- ESP_OK

### esp_err_t esp_mesh_set_network_duty_cycle(int nwk_duty, int duration_mins, int applied_rule)

Set the network duty cycle, duration and rule.

- The range of nwk_duty values is 1 to 100. The default value is 10.
- nwk_duty is the network duty cycle the entire network or the up-link path will use. A device that successfully sets the nwk_duty is known as a NWK-DUTY-MASTER.
- duration_mins specifies how long the specified nwk_duty will be used. Once duration_mins expires, the root will take over as the NWK-DUTY-MASTER. If an existing NWK-DUTY-MASTER leaves the network, the root will take over as the NWK-DUTY-MASTER again.
- duration_mins = (-1) represents nwk_duty will be used until a new NWK-DUTY-MASTER with a different nwk_duty appears.
- Only the root can set duration_mins to (-1).
- If applied_rule is set to MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE, the nwk_duty will be used by the entire network.
- If applied_rule is set to MESH_PS_NETWORK_DUTY_APPLIED_UPLINK, the nwk_duty will only be used by the up-link path nodes.
- The root does not accept MESH_PS_NETWORK_DUTY_APPLIED_UPLINK.
- A nwk_duty with duration_mins(-1) set by the root is the default network duty cycle used by the entire network.

**Attention** This API can be called at any time after mesh is started.
- In self-organized network, if this API is called before mesh is started in all devices, (1)nwk_duty shall be set to the same value for all devices; (2)duration_mins shall be set to (-1); (3)applied_rule shall be set to MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE; after the voted root appears, the root will become the NWK-DUTY-MASTER and broadcast the nwk_duty and its identity of NWK-DUTY-MASTER.
- If the root is specified (FIXED-ROOT), call this API in the root to provide a default nwk_duty for the entire network.
- After joins the network, any device can call this API to change the nwk_duty, duration_mins or applied_rule.

**Parameters**
- **nwk_duty** [in] network duty cycle
- **duration_mins** [in] duration (unit: minutes)
- **applied_rule** [in] only support MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE

**Returns**
- ESP_OK
- ESP_FAIL

### esp_err_t esp_mesh_get_network_duty_cycle(int *nwk_duty, int *duration_mins, int *dev_duty_type, int *applied_rule)

Get the network duty cycle, duration, type and rule.

**Parameters**
- **nwk_duty** [out] current network duty cycle
- **duration_mins** [out] the duration of current nwk_duty
- **dev_duty_type** [out] if it includes MESH_PSDEVICE_DUTY_MASTER, this device is the current NWK-DUTY-MASTER.
- **applied_rule** [out] MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE

**Returns**
- ESP_OK
int esp_mesh_get_running_active_duty_cycle(void)

Get the running active duty cycle.

- The running active duty cycle of the root is 100.
- If duty type is set to MESH_PS_DEVICE_DUTY_REQUEST, the running active duty cycle is nwk_duty provided by the network.
- If duty type is set to MESH_PS_DEVICE_DUTY_DEMAND, the running active duty cycle is dev_duty specified by the users.
- In a mesh network, devices are typically working with a certain duty-cycle (transmitting, receiving and sleep) to reduce the power consumption. The running active duty cycle decides the amount of awake time within a beacon interval. At each start of beacon interval, all devices wake up, broadcast beacons, and transmit packets if they do have pending packets for their parents or for their children. Note that Low-duty-cycle means devices may not be active in most of the time, the latency of data transmission might be greater.

Returns the running active duty cycle

esp_err_t esp_mesh_ps_duty_signaling(int fwd_times)

Duty signaling.

Parameters

fwd_times [in] the times of forwarding duty signaling packets

Returns

- ESP_OK

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
Chapter 2. API Reference

- `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

- `mesh_event_ps_duty_t ps_duty`
  - PS duty information
union `mesh_rc_config_t`

```c
#include <esp_mesh.h>
```

Vote address configuration.

**Public Members**

```c
int attempts
```

max vote attempts before a new root is elected automatically by mesh network. (min:15, 15 by default)

```c
mesh_addr_t rc_addr
```

a new root address specified by users for API `esp_mesh_waive_root()`

**Structures**

```c
struct mip_t
```

IP address and port.

**Public Members**

```c
esp_ip4_addr_t ip4
```

IP address

```c
uint16_t port
```

port

```c
struct mesh_event_channel_switch_t
```

Channel switch information.

**Public Members**

```c
uint8_t channel
```

new channel

```c
struct mesh_event_connected_t
```

Parent connected information.

**Public Members**

```c
wifi_event_sta_connected_t connected
```

parent information, same as Wi-Fi event SYSTEM_EVENT_STA_CONNECTED does

```c
uint16_t self_layer
```

layer

```c
uint8_t duty
```

parent duty
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

uint16_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

```c
uint8_t number
```
the number of APs scanned

```c
struct mesh_event_network_state_t
```
Network state information.

Public Members

```c
bool is_rootless
```
whether current mesh network has a root

```c
struct mesh_event_ps_duty_t
```
PS duty information.

Public Members

```c
uint8_t duty
```
parent or child duty

```c
mesh_event_child_connected_t child_connected
```
child info

```c
struct mesh_opt_t
```
Mesh option.

Public Members

```c
uint8_t type
```
option type

```c
uint16_t len
```
option length

```c
uint8_t* val
```
option value

```c
struct mesh_data_t
```
Mesh data for esp_mesh_send() and esp_mesh_recv()
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, default 6, max 10 = max_connection + non-mesh_max_connection max mesh connections

uint8_t nonmesh_max_connection
    max non-mesh connections
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

cryptofuncs_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_PS
   mesh PS is not specified as enable or disable

ESP_ERR_MESH_RECV_RELEASE
   release esp_mesh_recv_toDS

MESH_DATA_ENC
   Flags bitmap for esp_mesh_send() and esp_mesh_recv()
   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_recv()
   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_recv().

**MESH_ASSOC_FLAG_MAP_ASSOC**

Flag of mesh networking IE.

Mesh AP doesn’t detect children leave yet.

**MESH_ASSOC_FLAG_VOTE_IN_PROGRESS**

Station in vote, set when root vote start, clear when connect to router or when root switch.

**MESH_ASSOC_FLAG_STA_VOTED**

Station vote done, set when connect to router.

**MESH_ASSOC_FLAG_NETWORK_FREE**

No root in current network.

**MESH_ASSOC_FLAG_STA_VOTE_EXPIRE**

The voted address is expired, means the voted device lose the chance to be root.

**MESH_ASSOC_FLAG_ROOTS_FOUND**

Roots conflict is found, means that there are at least two roots in the mesh network.

**MESH_ASSOC_FLAG_ROOT_FIXED**

The root is fixed in the mesh network.

**MESH_PS_DEVICE_DUTY_REQUEST**

Mesh PS (Power Save) duty cycle type.

Requests to join a network PS without specifying a device duty cycle. After the device joins the network, a network duty cycle will be provided by the network.

**MESH_PS_DEVICE_DUTY_DEMAND**

Requests to join a network PS and specifies a demanded device duty cycle.

**MESH_PS_NETWORK_DUTY_MASTER**

Indicates the device is the NWK-DUTY-MASTER (network duty cycle master).

**MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE**

Mesh PS (Power Save) duty cycle applied rule.

**MESH_PS_NETWORK_DUTY_APPLIED_UPLINK**

**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

c enum mesh_event_id_t
Enumerated list of mesh event id.

Values:

enumerator MESH_EVENT_STARTED
mesh is started

enumerator MESH_EVENT_STOPPED
mesh is stopped

enumerator MESH_EVENT_CHANNEL_SWITCH
channel switch

enumerator MESH_EVENT_CHILD_CONNECTED
a child is connected on softAP interface

enumerator MESH_EVENT_CHILD_DISCONNECTED
a child is disconnected on softAP interface

enumerator MESH_EVENT_ROUTING_TABLE_ADD
routing table is changed by adding newly joined children

enumerator MESH_EVENT_ROUTING_TABLE_REMOVE
routing table is changed by removing leave children

enumerator MESH_EVENT_PARENT_CONNECTED
parent is connected on station interface

enumerator MESH_EVENT_PARENT_DISCONNECTED
parent is disconnected on station interface

enumerator MESH_EVENT_NO_PARENT_FOUND
no parent found
enumerator **MESH_EVENT_LAYER_CHANGE**
    layer changes over the mesh network

enumerator **MESH_EVENT_TODS_STATE**
    state represents whether the root is able to access external IP network. This state is a manual event that needs to be triggered with esp_mesh_post_toDS_state().

enumerator **MESH_EVENT_VOTE_STARTED**
    the process of voting a new root is started either by children or by the root

enumerator **MESH_EVENT_VOTE_STOPPED**
    the process of voting a new root is stopped

enumerator **MESH_EVENT_ROOT_ADDRESS**
    the root address is obtained. It is posted by mesh stack automatically.

enumerator **MESH_EVENT_ROOT_SWITCH_REQ**
    root switch request sent from a new voted root candidate

enumerator **MESH_EVENT_ROOT_SWITCH_ACK**
    root switch acknowledgment responds the above request sent from current root

enumerator **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.

enumerator **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.

enumerator **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.

enumerator **MESH_EVENT_NETWORK_STATE**
    network state, such as whether current mesh network has a root.

enumerator **MESH_EVENT_STOP_RECONNECTION**
    the root stops reconnecting to the router and non-root devices stop reconnecting to their parents.

enumerator **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.

enumerator **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.
enumerator **MESH_EVENT_PS_PARENT_DUTY**
    parent duty

enumerator **MESH_EVENT_PS_CHILD_DUTY**
    child duty

enumerator **MESH_EVENT_PS_DEVICE_DUTY**
    device duty

enumerator **MESH_EVENT_MAX**

**enum mesh_type_t**
    Device type.
    
    **Enumerator Values:**
    
    enumerator **MESH_IDLE**
        hasn’t joined the mesh network yet
    
    enumerator **MESH_ROOT**
        the only sink of the mesh network. Has the ability to access external IP network
    
    enumerator **MESH_NODE**
        intermediate device. Has the ability to forward packets over the mesh network
    
    enumerator **MESH_LEAF**
        has no forwarding ability
    
    enumerator **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.
    
    **Enumerator Values:**
    
    enumerator **MESH_PROTO_BIN**
        binary
    
    enumerator **MESH_PROTO_HTTP**
        HTTP protocol
    
    enumerator **MESH_PROTO_JSON**
        JSON format
    
    enumerator **MESH_PROTO_MQTT**
        MQTT protocol
    
    enumerator **MESH_PROTO_AP**
        IP network mesh communication of node’s AP interface
enumerator `MESH_PROTO_STA`
   IP network mesh communication of node’s STA interface

**enum** `mesh_tos_t`
For reliable transmission, mesh stack provides three type of services.

Values:

enumerator `MESH_TOS_P2P`
   provide P2P (point-to-point) retransmission on mesh stack by default

enumerator `MESH_TOS_E2E`
   provide E2E (end-to-end) retransmission on mesh stack (Unimplemented)

enumerator `MESH_TOS_DEF`
   no retransmission on mesh stack

**enum** `mesh_vote_reason_t`
Vote reason.

Values:

enumerator `MESH_VOTE_REASON_ROOT_INITIATED`
   vote is initiated by the root

enumerator `MESH_VOTE_REASON_CHILD_INITIATED`
   vote is initiated by children

**enum** `mesh_disconnect_reason_t`
Mesh disconnect reason code.

Values:

enumerator `MESH_REASON_CYCLIC`
   cyclic is detected

enumerator `MESH_REASON_PARENT_IDLE`
   parent is idle

enumerator `MESH_REASON_LEAF`
   the connected device is changed to a leaf

enumerator `MESH_REASON_DIFF_ID`
   in different mesh ID

enumerator `MESH_REASON_ROOTS`
   root conflict is detected

enumerator `MESH_REASON_PARENT_STOPPED`
   parent has stopped the mesh
enumerator **MESH_REASON_SCAN_FAIL**
scan fail

enumerator **MESH_REASON_IE_UNKNOWN**
unknown IE

enumerator **MESH_REASON_WAIVE_ROOT**
waive root

enumerator **MESH_REASON_PARENT_WORSE**
parent with very poor RSSI

enumerator **MESH_REASON_EMPTY_PASSWORD**
use an empty password to connect to an encrypted parent

enumerator **MESH_REASON_PARENT_UNENCRYPTED**
connect to an unencrypted parent/router

define **esp_mesh_topology_t**
Mesh topology.

Values:

enumerator **MESH_TOPO_TREE**
tree topology

enumerator **MESH_TOPO_CHAIN**
chain topology

define **mesh_event_toDS_state_t**
The reachability of the root to a DS (distribute system)

Values:

enumerator **MESH_TODS_UNREACHABLE**
the root isn’t able to access external IP network

enumerator **MESH_TODS_REACHABLE**
the root is able to access external IP network

**SmartConfig**

The SmartConfig™ is a provisioning technology developed by TI to connect a new Wi-Fi device to a Wi-Fi network. It uses a mobile application to broadcast the network credentials from a smartphone, or a tablet, to an un-provisioned Wi-Fi device.

The advantage of this technology is that the device does not need to directly know SSID or password of an Access Point (AP). This information is provided using the smartphone. This is particularly important to headless device and systems, due to their lack of a user interface.

If you are looking for other options to provision your ESP32-S2 devices, check Provisioning API.
**Application Example**  Connect ESP32-S2 to the target AP using SmartConfig: `wifi/smart_config`.

**API Reference**

**Header File**

- components/esp_wifi/include/esp_smartconfig.h
- This header file can be included with:
  ```c
  #include "esp_smartconfig.h"
  ```

This header file is a part of the API provided by the `esp_wifi` component. To declare that your component depends on `esp_wifi`, add the following to your CMakeLists.txt:

- `REQUIRES esp_wifi`
- `PRIV_REQUIRES esp_wifi`

**Functions**

- `const char* esp_smartconfig_get_version(void)`
  Get the version of SmartConfig.
  **Returns**
  - 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.
  **Parameters** `config` — pointer to smartconfig start configure structure
  **Returns**
  - ESP_OK: succeed
  - others: fail

- `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`.
  **Returns**
  - 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.
  **Parameters** `time_s` — range 15s~255s, offset: 45s.
Chapter 2. API Reference

**Returns**
- ESP_OK: succeed
- others: fail

`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.

**Parameters**
- `type` Choose from the smartconfig_type_t.

**Returns**
- ESP_OK: succeed
- others: fail

`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.
2. Fast mode have corresponding APP(phone).
3. Two mode is compatible.

**Parameters**
- `enable` false-disable(default); true-enable;

**Returns**
- ESP_OK: succeed
- others: fail

`esp_err_t esp_smartconfig_get_rvd_data (uint8_t* rvd_data, uint8_t len)`
Get reserved data of ESPTouch v2.

**Parameters**
- `rvd_data` reserved data
- `len` length of reserved data

**Returns**
- ESP_OK: succeed
- others: fail

**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.

bool esp_touch_v2_enable_crypt
    Enable ESPTouch v2 crypt.

char *esp_touch_v2_key
    ESPTouch v2 crypt key, len should be 16.

**Macros**
SMARTCONFIG_START_CONFIG_DEFAULT ()

**Enumerations**

enum *smartconfig_type_t*
    Values:

    enumerator SC_TYPE_ESPTOUCH
        protocol: ESPTouch

    enumerator SC_TYPE_AIRKISS
        protocol: AirKiss

    enumerator SC_TYPE_ESPTOUCH_AIRKISS
        protocol: ESPTouch and AirKiss

    enumerator SC_TYPE_ESPTOUCH_V2
        protocol: ESPTouch v2

enum *smartconfig_event_t*
    Smartconfig event declarations
    Values:
enumerator **SC_EVENT_SCAN_DONE**
Station smartconfig has finished to scan for APs

enumerator **SC_EVENT_FOUND_CHANNEL**
Station smartconfig has found the channel of the target AP

enumerator **SC_EVENT_GOT_SSID_PSWD**
Station smartconfig got the SSID and password

enumerator **SC_EVENT_SEND_ACK_DONE**
Station smartconfig has sent ACK to cellphone

**Wi-Fi**

**Introduction**  The Wi-Fi libraries provide support for configuring and monitoring the ESP32-S2 Wi-Fi networking functionality. This includes configuration for:

- Station mode (aka STA mode or Wi-Fi client mode). ESP32-S2 connects to an access point.
- AP mode (aka Soft-AP mode or Access Point mode). Stations connect to the ESP32-S2.
- Station/AP-coexistence mode (ESP32-S2 is concurrently an access point and a station connected to another access point).
- Various security modes for the above (WPA, WPA2, WPA3, etc.)
- Scanning for access points (active & passive scanning).
- Promiscuous mode for monitoring of IEEE802.11 Wi-Fi packets.

**Application Examples**  Several application examples demonstrating the functionality of Wi-Fi library are provided in the `wifi` directory of ESP-IDF repository. Please check the README for more details.

**API Reference**

**Header File**

- components/esp_wifi/include/esp_wifi.h
  - This header file can be included with:

    ```c
    #include "esp_wifi.h"
    ```

  - This header file is a part of the API provided by the `esp_wifi` component. To declare that your component depends on `esp_wifi`, add the following to your CMakeLists.txt:

    ```cmake
    REQUIRE esp_wifi
    ```

    or

    ```cmake
    PRIV_REQUIRE esp_wifi
    ```

**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!

**Parameters** config—pointer to WiFi initialized configuration structure; can point to a temporary variable.

**Returns**
- ESP_OK: succeed
- ESP_ERR_NO_MEM: out of memory
- others: refer to error code esp_err.h

```c
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

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

```c
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, station+soft-AP or NAN. The default mode is station mode.

**Parameters** mode—WiFi operating mode

**Returns**
- 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 esp_err.h

```c
esp_err_t esp_wifi_get_mode (wifi_mode_t *mode)
```

Get current operating mode of WiFi.

**Parameters** mode—[out] store current WiFi mode

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

```c
esp_err_t esp_wifi_start (void)
```

Start WiFi according to current configuration If mode is WIFI_MODE_STA, it creates station control block and starts station If mode is WIFI_MODE_AP, it creates soft-AP control block and starts soft-AP If mode is WIFI_MODE_APSTA, it creates soft-AP and station control block and starts soft-AP and station If mode is WIFI_MODE_NAN, it creates NAN control block and starts NAN.

**Returns**
- 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
**Chapter 2. API Reference**

- ESP_FAIL: other WiFi internal errors

**esp_err_t esp_wifi_stop (void)**

Stop WiFi. If mode is WIFI_MODE_STA, it stops station and frees station control block. If mode is WIFI_MODE_AP, it stops soft-AP and frees soft-AP control block. If mode is WIFI_MODE_APSTA, it stops station/soft-AP and frees station/soft-AP control block. If mode is WIFI_MODE_NAN, it stops NAN and frees NAN control block.

**Returns**
- 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

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

**esp_err_t esp_wifi_connect (void)**

Connect WiFi station to the AP.

**Attention**
1. This API only impact WIFI_MODE_STA or WIFI_MODE_APSTA mode
2. If station interface is connected to an AP, call esp_wifi_disconnect to disconnect.
3. The scanning triggered by esp_wifi_scan_start() will not be effective until connection between device and the AP is established. If device is scanning and connecting at the same time, it will abort scanning and return a warning message and error number ESP_ERR_WIFI_STATE.
4. This API attempts to connect to an Access Point (AP) only once. To enable reconnection in case of a connection failure, please use the ‘failure_retry_cnt’ feature in the ‘wifi_sta_config_t’. Users are suggested to implement reconnection logic in their application for scenarios where the specified AP does not exist, or reconnection is desired after the device has received a disconnect event.

**Returns**
- 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 WiFi station from the AP.

**Returns**
- 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.

**Returns**
- ESP_OK: succeed
Chapter 2. API Reference

- others: fail

```c
esp_err_t esp_wifi_deauth_sta(uint16_t aid)
```
deauthenticate all stations or associated id equals to aid

**Parameters**
- `aid` - when aid is 0, deauthenticate all stations, otherwise deauthenticate station whose associated id is aid

**Returns**
- 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

```c
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 memory 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.

**Parameters**
- `config` - configuration settings for scanning, if set to NULL default settings will be used of which default values are show_hidden:false, scan_type:active, scan_time.active.min:0, scan_time.active.max:120 miliseconds, scan_time.passive:360 miliseconds
- `block` - if block is true, this API will block the caller until the scan is done, otherwise it will return immediately

**Returns**
- 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

```c
esp_err_t esp_wifi_scan_stop(void)
```
Stop the scan in process.

**Returns**
- 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

```c
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.

**Parameters**
- `number` - [out] store number of APs found in last scan

**Returns**
- 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_t esp_wifi_scan_get_ap_records (uint16_t *number, wifi_ap_record_t *ap_records)

Get AP list found in last scan.

**Parameters**
- **number** - [inout] 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

**Returns**
- 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

### esp_err_t esp_wifi_clear_ap_list (void)

Clear AP list found in last scan.

**Attention**  When the obtained ap list fails, bss info must be cleared, otherwise it may cause memory leakage.

**Returns**
- 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_MODE: WiFi mode is wrong
- ESP_ERR_INVALID_ARG: invalid argument

### esp_err_t esp_wifi_sta_get_ap_info (wifi_ap_record_t *ap_info)

Get information of AP to which the device is associated with.

**Attention**  When the obtained country information is empty, it means that the AP does not carry country information

**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

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_CONN: The station interface don’t initialized
- ESP_ERR_WIFI_NOT_CONNECT: The station is in disconnect status

### 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.

**Parameters**
- **type** - power save type

**Returns**
- ESP_OK: succeed

### 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.

**Parameters**
- **type** - [out] store current power save type
Returns  ESP_OK: succeed

```c
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_PROTOCOL_11N). If `CONFIG_SOC_WIFI_HE_SUPPORT`, the default protocol is (WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOCOL_11N|WIFI_PROTOCOL_11AX).

Attention Support 802.11b or 802.11bg or 802.11bgn or 802.11bgnax or LR mode

Parameters
- `ifx` – interfaces
- `protocol_bitmap` – WiFi protocol bitmap

Returns
- 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

```c
esp_err_t esp_wifi_get_protocol(wifi_interface_t ifx, uint8_t* protocol_bitmap)
```

Get the current protocol bitmap of the specified interface.

Parameters
- `ifx` – interface
- `protocol_bitmap` – [out] store current WiFi protocol bitmap of interface `ifx`

Returns
- 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

```c
esp_err_t esp_wifi_set_bandwidth(wifi_interface_t ifx, wifi_bandwidth_t bw)
```

Set the bandwidth of 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

Parameters
- `ifx` – interface to be configured
- `bw` – bandwidth

Returns
- 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

```c
esp_err_t esp_wifi_get_bandwidth(wifi_interface_t ifx, wifi_bandwidth_t* bw)
```

Get the bandwidth of specified interface.

Attention 1. API return false if try to get a interface that is not enable

Parameters
- `ifx` – interface to be configured
- `bw` – [out] store bandwidth of interface `ifx`

Returns
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

\texttt{esp_err_t esp_wifi_set_channel} (uint8_t primary, wifi_second_chan_t second)

Set primary/secondary channel of device.

\textbf{Attention} 1. This API should be called after \texttt{esp_wifi_start()} and before \texttt{esp_wifi_stop()}
\textbf{Attention} 2. When device is in STA mode, this API should not be called when STA is scanning or connecting to an external AP
\textbf{Attention} 3. When device is in softAP mode, this API should not be called when softAP has connected to external STAs
\textbf{Attention} 4. When device is in STA+softAP mode, this API should not be called when in the scenarios described above
\textbf{Attention} 5. The channel info set by this API will not be stored in NVS. So If you want to remember the channel used before wifi stop, you need to call this API again after wifi start, or you can call \texttt{esp_wifi_set_config()} to store the channel info in NVS.

\textbf{Parameters}
- \texttt{primary} – for HT20, primary is the channel number, for HT40, primary is the primary channel
- \texttt{second} – for HT20, second is ignored, for HT40, second is the second channel

\textbf{Returns}
- 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
- ESP_ERR_WIFI_NOT_STARTED: WiFi is not started by esp_wifi_start

\texttt{esp_err_t esp_wifi_get_channel} (uint8_t *primary, wifi_second_chan_t *second)

Get the primary/secondary channel of device.

\textbf{Attention} 1. API return false if try to get a interface that is not enable

\textbf{Parameters}
- \texttt{primary} – store current primary channel
- \texttt{second} – [out] store current second channel

\textbf{Returns}
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

\texttt{esp_err_t esp_wifi_set_country} (const wifi_country_t *country)

configure country info

\textbf{Attention} 1. It is discouraged to call this API since this doesn’t validate the per-country rules, it’s up to the user to fill in all fields according to local regulations. Please use \texttt{esp_wifi_set_country_code} instead.
\textbf{Attention} 2. The default country is “01” (world safe mode) \{.cc=”US”, .schan=1, .nchan=11, .policy=WIFI_COUNTRY_POLICY_AUTO\}.
\textbf{Attention} 3. The third octet of country code string is one of the following: ‘1’, ‘O’, ‘I’, ‘X’, otherwise it is considered as ‘1’.
\textbf{Attention} 4. 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=”US”, .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 to the country info of the station automatically, {.cc=“US”,.schan=1,.nchan=11} in the example.

**Attention** 5. When the country policy is WIFI_COUNTRY_POLICY_MANUAL, then the configured country info is used always.

**Attention** 6. 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 also changed.

**Attention** 7. The country configuration is stored into flash.

**Attention** 8. When this API is called, the PHY init data will switch to the PHY init data type corresponding to the country info.

### Parameters

- **country**—the configured country info

### Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

```c
esp_err_t esp_wifi_get_country (wifi_country_t *country)
```

get the current country info

### Parameters

- **country**—country info

### Returns

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

```c
esp_err_t esp_wifi_set_mac (wifi_interface_t ifx, const uint8_t mac[6])
```

Set MAC address of WiFi station, soft-AP or NAN interface.

**Attention** 1. This API can only be called when the interface is disabled

**Attention** 2. Above mentioned interfaces have different MAC addresses, do not set them to be the same.

**Attention** 3. The bit 0 of the first byte of 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”.

### Parameters

- **ifx**—interface
- **mac**—the MAC address

### Returns

- 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

```c
esp_err_t esp_wifi_get_mac (wifi_interface_t ifx, uint8_t mac[6])
```

Get mac of specified interface.

### Parameters

- **ifx**—interface
- **mac**—[out] store mac of the interface ifx

### Returns

- 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_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.

**Parameters**

- cb – callback

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

**esp_err_t esp_wifi_set_promiscuous (bool en)**

Enable the promiscuous mode.

**Parameters**

- en – false - disable, true - enable

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

**esp_err_t esp_wifi_get_promiscuous (bool *en)**

Get the promiscuous mode.

**Parameters**

- en – [out] store the current status of promiscuous mode

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

**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

**Parameters**

- filter – the packet type filtered in promiscuous mode.

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

**esp_err_t esp_wifi_get_promiscuous_filter (wifi_promiscuous_filter_t *filter)**

Get the promiscuous filter.

**Parameters**

- filter – [out] store the current status of promiscuous filter

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

**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.

**Parameters**

- filter – the subtype of the control packet filtered in promiscuous mode.

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
**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.

**Parameters**
- `filter` - [out] store the current status of subtype filter of the control packet in promiscuous mode

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

**esp_err_t esp_wifi_set_config** *(wifi_interface_t interface, wifi_config_t *conf)*

Set the configuration of the STA, AP or NAN.

**Attention**
1. This API can be called only when specified interface is enabled, otherwise, API fail
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.
3. ESP devices are 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 station.
4. The configuration will be stored in NVS for station and soft-AP

**Parameters**
- `interface` - interface
- `conf` - station, soft-AP or NAN configuration

**Returns**
- 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 error code in esp_err.h

**esp_err_t esp_wifi_get_config** *(wifi_interface_t interface, wifi_config_t *conf)*

Get configuration of specified interface.

**Parameters**
- `interface` - interface
- `conf` - [out] station or soft-AP configuration

**Returns**
- 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_t esp_wifi_ap_get_sta_list** *(wifi_sta_list_t *sta)*

Get STAs associated with soft-AP.

**Attention**
SSC only API

**Parameters**
- `sta` - [out] station list ap can get the connected sta’s phy mode info through the struct member phy_11b, phy_11g, phy_11n, phy_lr in the wifi_sta_info_t struct. For example, phy_11b = 1 imply that sta support 802.11b mode

**Returns**
- 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

```c
esp_err_t esp_wifi_ap_get_sta_aid(const uint8_t* mac, uint16_t* aid)
```
Get AID of STA connected with soft-AP.

**Parameters**
- `mac` – STA’s mac address
- `aid` – [out] Store the AID corresponding to STA mac

**Returns**
- 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

```c
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

**Parameters**
- `storage` – storage type

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

```c
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.

**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 `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.

**Returns**
- 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

```c
esp_err_t esp_wifi_set_vendor_ie_cb(esp_vendor_ie_cb_t cb, void*ctx)
```
Register Vendor-Specific Information Element monitoring callback.

**Parameters**
- `cb` – Callback function
- `ctx` – Context argument, passed to callback function.

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
**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\}}.

**Parameters** power – Maximum WiFi transmitting power.

**Returns**
- 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, e.g. parameter is out of range

**esp_err_t esp_wifi_get_max_tx_power**(int8_t *power)

Get maximum transmitting power after WiFi start.

**Parameters** power – Maximum WiFi transmitting power, unit is 0.25dBm.

**Returns**
- 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_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_PROBEREQRECVED.
**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.

**Parameters** mask – WiFi event mask.

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

**esp_err_t esp_wifi_get_event_mask**(uint32_t *mask)

Get mask of WiFi events.

**Parameters** mask – WiFi event mask.

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

**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

**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_INVALID_ARG is returned.

**Returns**
- ESP_OK: success
- ESP_ERR_WIFI_IF: Invalid interface
- ESP_ERR_INVALID_ARG: Invalid parameter
- ESP_ERR_WIFI_NO_MEM: out of memory

```c
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.

**Parameters**
- **cb** – callback
- **ctx** – context argument, passed to callback function

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init

```c
esp_err_t esp_wifi_set_csi_config (const wifi_csi_config_t *config)
```

Set CSI data configuration.

**Parameters**
- **config** – configuration

**Returns**
- 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 or promiscuous mode is not enabled
- ESP_ERR_INVALID_ARG: invalid argument

```c
esp_err_t esp_wifi_set_csi (bool en)
```

Enable or disable CSI.

**Returns**
- 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 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.

**Parameters**
config – Antenna GPIO configuration.

**Returns**
• ESP_OK: succeed
• ESP_ERR_WIFI_NOT_STARTED: WiFi is not initialized by esp_wifi_init
• ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is NULL, invalid GPIO number etc

*esp_err_t* **esp_wifi_get_ant_gpio** (wifi_ant_gpio_config_t *config*)

Get current antenna GPIO configuration.

**Parameters**
config – Antenna GPIO configuration.

**Returns**
• ESP_OK: succeed
• ESP_ERR_WIFI_NOT_STARTED: WiFi is not initialized by esp_wifi_init
• ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is NULL

*esp_err_t* **esp_wifi_set_ant** (const *wifi_ant_config_t* *config*)

Set antenna configuration.

**Parameters**
config – Antenna configuration.

**Returns**
• ESP_OK: succeed
• ESP_ERR_WIFI_NOT_STARTED: WiFi is not initialized by esp_wifi_init
• ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is NULL, invalid antenna mode or invalid GPIO number

*esp_err_t* **esp_wifi_get_ant** (wifi_ant_config_t *config*)

Get current antenna configuration.

**Parameters**
config – Antenna configuration.

**Returns**
• ESP_OK: succeed
• ESP_ERR_WIFI_NOT_STARTED: WiFi is not initialized by esp_wifi_init
• ESP_ERR_INVALID_ARG: invalid argument, e.g. parameter is NULL

*int64_t* **esp_wifi_get_tsf_time** (wifi_interface_t interface)

Get the TSF time In Station mode or SoftAP+Station mode if station is not connected or station doesn’t receive at least one beacon after connected, will return 0.

**Attention** Enabling power save may cause the return value inaccurate, except WiFi modem sleep

**Parameters**
interface - The interface whose tsf_time is to be retrieved.

**Returns**
0 or the TSF time

*esp_err_t* **esp_wifi_set_inactive_time** (wifi_interface_t ifx, uint16_t sec)

Set the inactive time of the 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 deauth the STA. Default is 300s.

**Attention** 3. The inactive time configuration is not stored into flash.

### Parameters
- `ifx` – interface to be configured.
- `sec` – Inactive time. Unit seconds.

### Returns
- `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, For Station, if `sec` is less than 3. For SoftAP, if `sec` is less than 10.

```c
esp_err_t esp_wifi_get_inactive_time(wifi_interface_t ifx, uint16_t* sec)
```

Get inactive time of specified interface.

### Parameters
- `ifx` – Interface to be configured.
- `sec` – Inactive time. Unit seconds.

### Returns
- `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

```c
esp_err_t esp_wifi_set_rssi_threshold(int32_t rssi)
```

Set RSSI threshold, if average rssi gets lower than threshold, WiFi task will post event WIFI_EVENT_STA_BSS_RSSI_LOW.

### Parameters
- `rssi` – threshold value in dbm between -100 to 10. Note that in some rare cases where signal strength is very strong, rssi values can be slightly positive.

### Returns
- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by esp_wifi_init
- `ESP_ERR_INVALID_ARG`: invalid argument

```c
esp_err_t esp_wifi_ftm_initiate_session(wifi_ftm_initiator_cfg_t *cfg)
```

Start an FTM Initiator session by sending FTM request. If successful, event WIFI_EVENT_FTM_REPORT is generated with the result of the FTM procedure.

### Attention
1. Use this API only in Station mode.
2. If FTM is initiated on a different channel than Station is connected in or internal SoftAP is started in, FTM defaults to a single burst in ASAP mode.

### Parameters
- `cfg` – FTM Initiator session configuration
Returns

- ESP_OK: succeed
- others: failed

`esp_err_t esp_wifi_ftm_end_session(void)`
End the ongoing FTM Initiator session.

**Attention**  This API works only on FTM Initiator

Returns

- ESP_OK: succeed
- others: failed

`esp_err_t esp_wifi_ftm_resp_set_offset(int16_t offset_cm)`
Set offset in cm for FTM Responder. An equivalent offset is calculated in picoseconds and added in TOD of FTM Measurement frame (T1).

**Attention**  Use this API only in AP mode before performing FTM as responder

**Parameters**
offset_cm – T1 Offset to be added in centimeters

**Returns**

- ESP_OK: succeed
- others: failed

`esp_err_t esp_wifi_config_11b_rate(wifi_interface_t ifx, bool disable)`
Enable or disable 11b rate of specified interface.

**Attention** 1. This API should be called after esp_wifi_init() and before esp_wifi_start().
**Attention** 2. Only when really need to disable 11b rate call this API otherwise don’t call this.

**Parameters**

- ifx – Interface to be configured.
- disable – true means disable 11b rate while false means enable 11b rate.

**Returns**

- ESP_OK: succeed
- others: failed

`esp_err_t esp_wifi_connectionless_module_set_wake_interval(uint16_t wake_interval)`
Set wake interval for connectionless modules to wake up periodically.

**Attention** 1. Only one wake interval for all connectionless modules.
**Attention** 2. This configuration could work at connected status. When `ESP_WIFI_STA_DISCONNECTED_PM_ENABLE` is enabled, this configuration could work at disconnected status.
**Attention** 3. Event `WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START` would be posted each time wake interval starts.
**Attention** 4. Recommend to configure interval in multiples of hundred. (e.g. 100ms)
**Attention** 5. Recommend to configure interval to `ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE` to get stable performance at coexistence mode.

**Parameters**

- wake_interval – Milliseconds after would the chip wake up, from 1 to 65535.
**esp_err_t** `esp_wifi_force_wakeup_acquire(void)`

Request extra reference of Wi-Fi radio. Wi-Fi keep active state (RF opened) to be able to receive packets.

**Attention** Please pair the use of `esp_wifi_force_wakeup_acquire` with `esp_wifi_force_wakeup_release`.

**Returns**
- 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_force_wakeup_release(void)`

Release extra reference of Wi-Fi radio. Wi-Fi goes to sleep state (RF closed) if no more use of radio.

**Attention** Please pair the use of `esp_wifi_force_wakeup_acquire` with `esp_wifi_force_wakeup_release`.

**Returns**
- 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_set_country_code(const char *country, bool ieee80211d_enabled)`

Configure country

**Attention** 1. When `ieee80211d_enabled` is enabled, the country info of the AP to which the station is connected is used. E.g. if the configured country is US and the country info of the AP to which the station is connected is JP then the country info that will be used is JP. If the station disconnected from the AP the country info is set back to the country info of the station automatically, US in the example.

**Attention** 2. When `ieee80211d_enabled` is disabled, then the configured country info is used always.

**Attention** 3. 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 also changed.

**Attention** 4. The country configuration is stored into flash.

**Attention** 5. When this API is called, the PHY init data will switch to the PHY init data type corresponding to the country info.


**Attention** 7. When country code “01” (world safe mode) is set. SoftAP mode won’t contain country IE.

**Attention** 8. The default country is “01” (world safe mode) and `ieee80211d_enabled` is TRUE.

**Attention** 9. The third octet of country code string is one of the following: ‘‘, ‘O’, ‘I’, ‘X’, otherwise it is considered as ‘‘.

**Parameters**
- `country` – the configured country ISO code
- `ieee80211d_enabled` – 802.11d is enabled or not

**Returns**
- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by `esp_wifi_init`
- ESP_ERR_INVALID_ARG: invalid argument
**esp_err_t esp_wifi_get_country_code (char *country)**

get the current country code

**Parameters**

- country – country code

**Returns**

- ESP_OK: succeed
- ESP_ERR_WIFI_NOT_INIT: WiFi is not initialized by esp_wifi_init
- ESP_ERR_INVALID_ARG: invalid argument

**esp_err_t esp_wifi_config_80211_tx_rate (wifi_interface_t ifx, wifi_phy_rate_t rate)**

Config 80211 tx rate of specified interface.

**Attention** 1. This API should be called after esp_wifi_init() and before esp_wifi_start().

**Parameters**

- ifx – Interface to be configured.
- rate – Phy rate to be configured.

**Returns**

- ESP_OK: succeed
- others: failed

**esp_err_t esp_wifi_disable_pmf_config (wifi_interface_t ifx)**

Disable PMF configuration for specified interface.

**Attention** This API should be called after esp_wifi_set_config() and before esp_wifi_start().

**Parameters**

- ifx – Interface to be configured.

**Returns**

- ESP_OK: succeed
- others: failed

**esp_err_t esp_wifi_sta_get_aid (uint16_t *aid)**

Get the Association id assigned to STA by AP.

**Attention** aid = 0 if station is not connected to AP.

**Parameters**

- aid – [out] store the aid

**Returns**

- ESP_OK: succeed

**esp_err_t esp_wifi_sta_get_negotiated_phymode (wifi_phy_mode_t *phymode)**

Get the negotiated phymode after connection.

**Parameters**

- phymode – [out] store the negotiated phymode.

**Returns**

- ESP_OK: succeed

**esp_err_t esp_wifi_set_dynamic_cs (bool enabled)**

Config dynamic carrier sense.

**Attention** This API should be called after esp_wifi_start().

**Parameters**

- enabled – Dynamic carrier sense is enabled or not.

**Returns**

- ESP_OK: succeed
• others: failed

```c
esp_err_t esp_wifi_sta_get_rssi(int *rssi)
```

Get the rssi info after station connected to AP.

**Attention** This API should be called after station connected to AP.

**Parameters**
- `rssi` -- store the rssi info received from last beacon.

**Returns**
- ESP_OK: succeed
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_FAIL: failed

**Structures**

```c
struct wifi_init_config_t
```

WiFi stack configuration parameters passed to esp_wifi_init call.

**Public Members**

- `wifi_osi_funcs_t *osifuncs`
  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 rx_mgmt_buf_type`
  WiFi RX MGMT buffer type

- `int rx_mgmt_buf_num`
  WiFi RX MGMT buffer number

- `int cache_tx_buf_num`
  WiFi TX cache buffer number
int \texttt{csi\_enable}
\begin{itemize}
  \item WiFi channel state information enable flag
\end{itemize}

int \texttt{ampdu\_rx\_enable}
\begin{itemize}
  \item WiFi AMPDU RX feature enable flag
\end{itemize}

int \texttt{ampdu\_tx\_enable}
\begin{itemize}
  \item WiFi AMPDU TX feature enable flag
\end{itemize}

int \texttt{amsdu\_tx\_enable}
\begin{itemize}
  \item WiFi AMSDU TX feature enable flag
\end{itemize}

int \texttt{nvs\_enable}
\begin{itemize}
  \item WiFi NVS flash enable flag
\end{itemize}

int \texttt{nano\_enable}
\begin{itemize}
  \item Nano option for printf/scan family enable flag
\end{itemize}

int \texttt{rx\_ba\_win}
\begin{itemize}
  \item WiFi Block Ack RX window size
\end{itemize}

int \texttt{wifi\_task\_core\_id}
\begin{itemize}
  \item WiFi Task Core ID
\end{itemize}

int \texttt{beacon\_max\_len}
\begin{itemize}
  \item WiFi softAP maximum length of the beacon
\end{itemize}

int \texttt{mgmt\_sbuf\_num}
\begin{itemize}
  \item WiFi management short buffer number, the minimum value is 6, the maximum value is 32
\end{itemize}

uint64_t \texttt{feature\_caps}
\begin{itemize}
  \item Enables additional WiFi features and capabilities
\end{itemize}

bool \texttt{sta\_disconnected\_pm}
\begin{itemize}
  \item WiFi Power Management for station at disconnected status
\end{itemize}

int \texttt{espnow\_max\_encrypt\_num}
\begin{itemize}
  \item Maximum encrypt number of peers supported by espnow
\end{itemize}

int \texttt{magic}
\begin{itemize}
  \item WiFi init magic number, it should be the last field
\end{itemize}

Macros

\texttt{ESP\_ERR\_WIFI\_NOT\_INIT}
\begin{itemize}
  \item WiFi driver was not installed by \texttt{esp\_wifi\_init}
\end{itemize}

\texttt{ESP\_ERR\_WIFI\_NOT\_STARTED}
\begin{itemize}
  \item WiFi driver was not started by \texttt{esp\_wifi\_start}
\end{itemize}
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
Chapter 2. API Reference

**ESP_ERR_WIFI_NOT_ASSOC**

The WiFi connection is not associated

**ESP_ERR_WIFI_TX_DISALLOW**

The WiFi TX is disallowed

**ESP_ERR_WIFI_TWT_FULL**

no available flow id

**ESP_ERR_WIFI_TWT_SETUP_TIMEOUT**

Timeout of receiving twt setup response frame, timeout times can be set during twt setup

**ESP_ERR_WIFI_TWT_SETUP_TXFAIL**

TWT setup frame tx failed

**ESP_ERR_WIFI_TWT_SETUP_REJECT**

The twt setup request was rejected by the AP

**ESP_ERR_WIFI_DISCARD**

Discard frame

**WIFI_STATIC_TX_BUFFER_NUM**

**WIFI_CACHE_TX_BUFFER_NUM**

**WIFI_DYNAMIC_TX_BUFFER_NUM**

**WIFI_RX_MGMT_BUF_NUM_DEF**

**WIFI_CSI_ENABLED**

**WIFI_AMPDU_TX_ENABLED**

**WIFI_AMSDU_TX_ENABLED**

**WIFI_NVS_ENABLED**

**WIFI_NANO_FORMAT_ENABLED**

**WIFI_INIT_CONFIG_MAGIC**

**WIFI_DEFAULT_RX_BA_WIN**

**WIFI_TASK_CORE_ID**
WIFI_SOFTAP_BEACON_MAX_LEN

WIFI_MGMT_SBUF_NUM

WIFI_STA_DISCONNECTED_PM_ENABLED

CONFIG_FEATURE_WPA3_SAE_BIT

CONFIG_FEATURE_CACHE_TX_BUF_BIT

CONFIG_FEATURE_FTM_INITIATOR_BIT

CONFIG_FEATURE_FTM_RESPONDER_BIT

WIFI_INIT_CONFIG_DEFAULT()

ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE

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.

  **Param buf**  Data received. Type of data in buffer (wifi_promiscuous_pkt_t or wifi_pkt_rx_ctrl_t) indicated by `type` parameter.
  **Param 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.

  **Param ctx**  Context argument, as passed to esp_wifi_set_vendor_ie_cb() when registering callback.
  **Param type**  Information element type, based on frame type received.
  **Param sa**  Source 802.11 address.
  **Param vnd_ie**  Pointer to the vendor specific element data received.
  **Param 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.

  **Param ctx**  context argument, passed to esp_wifi_set_csi_rx_cb() when registering callback function.
  **Param data**  CSI data received. The memory that it points to will be deallocated after callback function returns.
Header File

- `components/esp_wifi/include/esp_wifi_types.h`
- This header file can be included with:
  ```
  #include "esp_wifi_types.h"
  ```
- This header file is a part of the API provided by the `esp_wifi` component. To declare that your component depends on `esp_wifi`, add the following to your `CMakeLists.txt`:
  ```
  REQUIRES esp_wifi
  or
  PRIV_REQUIRES esp_wifi
  ```

Unions

`union wifi_config_t`

```
#include <esp_wifi_types.h> Configuration data for device’s AP or STA or NAN.
```

The usage of this union (for ap, sta or nan 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

wifi_nan_config_t nan
configuration of NAN
```

Structures

`struct wifi_country_t`

Structure describing WiFi country-based regional restrictions.

Public Members

```
cchar 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, active or passive

**wifi_scan_time_t**

scan time per channel

uint8_t **home_chan_dwell_time**

time spent at home channel between scanning consecutive channels.

struct **wifi_he_ap_info_t**

Description of a WiFi AP HE Info.

**Public Members**

uint8_t **bss_color**

an unsigned integer whose value is the BSS Color of the BSS corresponding to the AP

uint8_t **partial_bss_color**

indicate if an AID assignment rule based on the BSS color

uint8_t **bss_color_disabled**

indicate if the use of BSS color is disabled

uint8_t **bssid_index**

in M-BSSID set, identifies the nontransmitted BSSID

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**

secondary channel of AP

int8_t **rssi**

signal strength of AP. Note that in some rare cases where signal strength is very strong, rssi values can be slightly positive
**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
bit: 0 flag to identify if 11b mode is enabled or not

**uint32_t** phy_11g
bit: 1 flag to identify if 11g mode is enabled or not

**uint32_t** phy_11n
bit: 2 flag to identify if 11n mode is enabled or not

**uint32_t** phy_lr
bit: 3 flag to identify if low rate is enabled or not

**uint32_t** phy_11ax
bit: 4 flag to identify if 11ax mode is enabled or not

**uint32_t** wps
bit: 5 flag to identify if WPS is supported or not

**uint32_t** ftm_responder
bit: 6 flag to identify if FTM is supported in responder mode

**uint32_t** ftm_initiator
bit: 7 flag to identify if FTM is supported in initiator mode

**uint32_t** reserved
bit: 8..31 reserved

**wifi_country_t** country
country information of AP

**wifi_he_ap_info_t** he_ap
HE AP info

**struct wifi_scan_threshold_t**
Structure describing parameters for a WiFi fast scan.
Public Members

```
int8_t rssi
```

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 Note: Incase this value is not set and password is set as per WPA2 standards(password len >= 8), it will be defaulted to WPA2 and device won’t connect to deprecated WEP/WPA networks. Please set authmode threshold as WIFI_AUTH_WEP/WIFI_AUTH_WPA_PSK to connect to WEP/WPA networks

Public Members

```
bool capable
```

Deprecated variable. Device will always 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.

Public Members

```
uint8_t ssid[32]
```

SSID of 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 soft-AP.

```
uint8_t ssid_len
```

Optional length of SSID field.

```
uint8_t channel
```

Channel of soft-AP

```
wifi_auth_mode_t authmode
```

Auth mode of soft-AP. Do not support AUTH_WEP, AUTH_WAPI_PSK and AUTH_OWE in soft-AP mode. When the auth mode is set to WPA2_PSK, WPA2_WPA3_PSK or WPA3_PSK, the pairwise cipher will be overwritten with WIFI_CIPHER_TYPE_CCMP.

```
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

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

**wifi_cipher_type_t** **pairwise_cipher**

Pairwise cipher of SoftAP, group cipher will be derived using this. Cipher values are valid starting from WIFI_CIPHER_TYPE_TKIP, enum values before that will be considered as invalid and default cipher suites (TKIP+CCMP) will be used. Valid cipher suites in softAP mode are WIFI_CIPHER_TYPE_TKIP, WIFI_CIPHER_TYPE_CCMP and WIFI_CIPHER_TYPE_TKIP_CCMP.

bool **ftm_responder**

Enable FTMResponder mode

**wifi_pmf_config_t** **pmf_cfg**

Configuration for Protected Management Frame

**wifi_sae_pwe_method_t** **sae_pwe_h2e**

Configuration for SAE PWE derivation method

struct **wifi_sta_config_t**

STA configuration settings for the device.

**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.
Chapter 2. API Reference

```c
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 scan_threshold 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.

uint32_t rm_enabled
  Whether Radio Measurements are enabled for the connection

uint32_t btm_enabled
  Whether BSS Transition Management is enabled for the connection

uint32_t mbo_enabled
  Whether MBO is enabled for the connection

uint32_t ft_enabled
  Whether FT is enabled for the connection

uint32_t owe_enabled
  Whether OWE is enabled for the connection

uint32_t transition_disable
  Whether to enable transition disable feature

uint32_t reserved
  Reserved for future feature set

wifi_sae_pwe_method_t sae_pwe_h2e
  Configuration for SAE PWE derivation method

wifi_sae_pk_mode_t sae_pk_mode
  Configuration for SAE-PK (Public Key) Authentication method

uint8_t failure_retry_cnt
  Number of connection retries station will do before moving to next AP. scan_method should be set as WIFI_ALL_CHANNEL_SCAN to use this config. Note: Enabling this may cause connection time to increase incase best AP doesn’t behave properly.

uint32_t he_dcm_set
  Whether DCM max.constellation for transmission and reception is set.
```
Chapter 2. API Reference

```
uint32_t he_dcm_max_constellation_tx
    Indicate the max.constellation for DCM in TB PPDU the STA supported. 0: not supported. 1: BPSK,
    2: QPSK, 3: 16-QAM. The default value is 3.

uint32_t he_dcm_max_constellation_rx
    Indicate the max.constellation for DCM in both Data field and HE-SIG-B field the STA supported. 0:
    not supported. 1: BPSK, 2: QPSK, 3: 16-QAM. The default value is 3.

uint32_t he_mcs9_enabled
    Whether to support HE-MCS 0 to 9. The default value is 0.

uint32_t he_su_beamformee_disabled
    Whether to disable support for operation as an SU beamformee.

uint32_t he_trig_su_bmforming_feedback_disabled
    Whether to disable support the transmission of SU feedback in an HE TB sounding sequence.

uint32_t he_trig_mu_bmforming_partial_feedback_disabled
    Whether to disable support the transmission of partial-bandwidth MU feedback in an HE TB sounding
    sequence.

uint32_t he_trig_cqi_feedback_disabled
    Whether to disable support the transmission of CQI feedback in an HE TB sounding sequence.

uint32_t he_reserved
    Reserved for future feature set

uint8_t sae_h2e_identifier[SAE_H2E_IDENTIFIER_LEN]
    Password identifier for H2E. this needs to be null terminated string

struct wifi_nan_config_t
    NAN Discovery start configuration.

Public Members

uint8_t op_channel
    NAN Discovery operating channel

uint8_t master_pref
    Device’s preference value to serve as NAN Master

uint8_t scan_time
    Scan time in seconds while searching for a NAN cluster

uint16_t warm_up_sec
    Warm up time before assuming NAN Anchor Master role

struct wifi_sta_info_t
    Description of STA associated with AP.
**Public Members**

```c
uint8_t mac[6]
    mac address

int8_t rssi
    current average rssi of sta connected

uint32_t phy_11b
    bit: 0 flag to identify if 11b mode is enabled or not

uint32_t phy_11g
    bit: 1 flag to identify if 11g mode is enabled or not

uint32_t phy_11n
    bit: 2 flag to identify if 11n mode is enabled or not

uint32_t phy_lr
    bit: 3 flag to identify if low rate is enabled or not

uint32_t phy_11ax
    bit: 4 flag to identify if 11ax mode is enabled or not

uint32_t is_mesh_child
    bit: 5 flag to identify mesh child

uint32_t reserved
    bit: 6..31 reserved
```

**Public Members**

```c
struct wifi_sta_list_t
    List of stations associated with the Soft-AP.

```

**Public Members**

```c
wifi_sta_info_t sta[ESP_WIFI_MAX_CONN_NUM]
    station list

int num
    number of stations in the list (other entries are invalid)
```

**Public Members**

```c
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
Received Signal Strength Indicator (RSSI) of packet. Unit: dBm

unsigned rate
PHY rate encoding of the packet. Only valid for non HT(11bg) packet

unsigned __pad0__
reserved

unsigned sig_mode
Protocol of the received packet, 0: non HT(11bg) packet; 1: HT(11n) packet; 3: VHT(11ac) packet

unsigned __pad1__
reserved

unsigned mcs
Modulation Coding Scheme. If is HT(11n) packet, shows the modulation, range from 0 to 76(MSC0 ~ MCS76)

unsigned cwb
Channel Bandwidth of the packet. 0: 20MHz; 1: 40MHz

unsigned __pad2__
reserved

unsigned smoothing
Set to 1 indicates that channel estimate smoothing is recommended. Set to 0 indicates that only per-carrier independent (unsmoothed) channel estimate is recommended.
unsigned **not_sounding**
Set to 0 indicates that PPDU is a sounding PPDU. Set to 1 indicates that the PPDU is not a sounding PPDU. Sounding PPDU is used for channel estimation by the request receiver

unsigned **__pad3__**
reserved

unsigned **aggregation**
Aggregation. 0: MPDU packet; 1: AMPDU packet

unsigned **stbc**
Space Time Block Code(STBC). 0: non STBC packet; 1: STBC packet

unsigned **fec_coding**
Forward Error Correction(FEC). Flag is set for 11n packets which are LDPC

unsigned **sgi**
Short Guide Interval(SGI). 0: Long GI; 1: Short GI

unsigned **__pad4__**
reserved

unsigned **ampdu_cnt**
the number of subframes aggregated in AMPDU

unsigned **channel**
primary channel on which this packet is received

unsigned **secondary_channel**
secondary channel on which this packet is received. 0: none; 1: above; 2: below

unsigned **__pad5__**
reserved

unsigned **timestamp**
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 **__pad6__**
reserved

unsigned **__pad7__**
reserved

unsigned **__pad8__**
reserved

unsigned **ant**
antenna number from which this packet is received. 0: WiFi antenna 0; 1: WiFi antenna 1
signed noise_floor
    noise floor of Radio Frequency Module(RF). unit: dBm

unsigned __pad9__
    reserved

unsigned sig_len
    length of packet including Frame Check Sequence(FCS)

unsigned __pad10__
    reserved

unsigned rx_state
    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 htlf_en
        enable to receive HT long training field(htlf) 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

uint8_t `dmac[6]`

   destination MAC address of the CSI data

bool `first_word_invalid`

   first four bytes of the CSI data is invalid or not, true indicates the first four bytes is invalid due to hardware limition

int8_t * `buf`

   valid buffer of CSI data

uint16_t `len`

   valid length of CSI data

uint8_t * `hdr`

   header of the wifi packet

uint8_t * `payload`

   payload of the wifi packet
Chapter 2. API Reference

uint16_t `payload_len`
payload len of the wifi packet

struct `wifi_ant_gpio_t`
WiFi GPIO configuration for antenna selection.

**Public Members**

uint8_t `gpio_select`
Whether this GPIO is connected to external antenna switch

uint8_t `gpio_num`
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`
Index (in antenna GPIO configuration) of enabled `WIFI_ANT_MODE_ANT0`

uint8_t `enabled_ant1`
Index (in antenna GPIO configuration) of enabled `WIFI_ANT_MODE_ANT1`

struct `wifi_action_tx_req_t`
Action Frame Tx Request.
Chapter 2. API Reference

Public Members

`wifi_interface_t ifx`
WiFi interface to send request to

`uint8_t dest_mac[6]`
Destination MAC address

`bool no_ack`
Indicates no ack required

`wifi_action_rx_cb_t rx_cb`
Rx Callback to receive any response

`uint32_t data_len`
Length of the appended Data

`uint8_t data[0]`
Appended Data payload

`struct wifi_ftm_initiator_cfg_t`
FTM Initiator configuration.

Public Members

`uint8_t resp_mac[6]`
MAC address of the FTM Responder

`uint8_t channel`
Primary channel of the FTM Responder

`uint8_t frm_count`
No. of FTM frames requested in terms of 4 or 8 bursts (allowed values - 0(No pref), 16, 24, 32, 64)

`uint16_t burst_period`
Requested time period between consecutive FTM bursts in 100’s of milliseconds (0 - No pref)

`struct wifi_beacon_monitor_config_t`
WiFi beacon monitor parameter configuration.

Public Members

`bool enable`
Enable or disable beacon monitor

`uint8_t loss_timeout`
Beacon lost timeout
Chapter 2. API Reference

- **uint8_t loss_threshold**
  - Maximum number of consecutive lost beacons allowed

- **uint8_t delta_intr_early**
  - Delta early time for RF PHY on

- **uint8_t delta_loss_timeout**
  - Delta timeout time for RF PHY off

**struct wifi_nan_publish_cfg_t**

NAN Publish service configuration parameters.

**Public Members**

- **char service_name[ESP_WIFI_MAX_SVC_NAME_LEN]**
  - Service name identifier

- **wifi_nan_service_type_t type**
  - Service type

- **char matching_filter[ESP_WIFI_MAX_FILTER_LEN]**
  - Comma separated filters for filtering services

- **char svc_info[ESP_WIFI_MAX_SVC_INFO_LEN]**
  - Service info shared in Publish frame

- **uint8_t single_replied_event**
  - Give single Replied event or every time

- **uint8_t datapath_reqd**
  - NAN Datapath required for the service

- **uint8_t reserved**
  - Reserved

**struct wifi_nan_subscribe_cfg_t**

NAN Subscribe service configuration parameters.

**Public Members**

- **char service_name[ESP_WIFI_MAX_SVC_NAME_LEN]**
  - Service name identifier

- **wifi_nan_service_type_t type**
  - Service type
char \texttt{matching\_filter}[\texttt{ESP\_WIFI\_MAX\_FILTER\_LEN}]
Comma separated filters for filtering services

char \texttt{svc\_info}[\texttt{ESP\_WIFI\_MAX\_SVC\_INFO\_LEN}]
Service info shared in Subscribe frame

uint8_t \texttt{single\_match\_event}
Give single Match event or every time

uint8_t \texttt{reserved}
Reserved

struct \texttt{wifi\_nan\_followup\_params\_t}
NAN Follow-up parameters.

**Public Members**

uint8_t \texttt{inst\_id}
Own service instance id

uint8_t \texttt{peer\_inst\_id}
Peer’s service instance id

uint8_t \texttt{peer\_mac}[6]
Peer’s MAC address

char \texttt{svc\_info}[\texttt{ESP\_WIFI\_MAX\_SVC\_INFO\_LEN}]
Service info(or message) to be shared

struct \texttt{wifi\_nan\_datapath\_req\_t}
NAN Datapath Request parameters.

**Public Members**

uint8_t \texttt{pub\_id}
Publisher’s service instance id

uint8_t \texttt{peer\_mac}[6]
Peer’s MAC address

bool \texttt{confirm\_required}
NDP Confirm frame required

struct \texttt{wifi\_nan\_datapath\_resp\_t}
NAN Datapath Response parameters.
Public Members

bool accept
    True - Accept incoming NDP, False - Reject it

uint8_t ndp_id
    NAN Datapath Identifier

uint8_t peer_mac[6]
    Peer’s MAC address

struct wifi_nan_datapath_end_req_t
    NAN Datapath End parameters.

Public Members

uint8_t ndp_id
    NAN Datapath Identifier

uint8_t peer_mac[6]
    Peer’s MAC address

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
Chapter 2. API Reference

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

uint16_t aid
authentication id assigned by the connected 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

int8_t rssi
rssiof 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 \texttt{pin\_code}[8]
PIN code of station in enrollee mode

struct \texttt{wifi\_event\_sta\_wps\_er\_success\_t}
Argument structure for WIFI_EVENT_STA_WPS_ER_SUCCESS event

Public Members

uint8_t \texttt{ap\_cred\_cnt}
Number of AP credentials received

uint8_t \texttt{ssid}[\text{MAX\_SSID\_LEN}]
SSID of AP

uint8_t \texttt{passphrase}[\text{MAX\_PASSPHRASE\_LEN}]
Passphrase for the AP

struct \texttt{wifi\_event\_sta\_wps\_er\_success\_t}::[anonymous] \texttt{ap\_cred}[\text{MAX\_WPS\_AP\_CRED}]
All AP credentials received from WPS handshake

struct \texttt{wifi\_event\_ap\_sta\_connected\_t}
Argument structure for WIFI_EVENT_AP_STACONNECTED event

Public Members

uint8_t \texttt{mac}[6]
MAC address of the station connected to Soft-AP

uint8_t \texttt{aid}
the aid that soft-AP gives to the station connected to

bool \texttt{is\_mesh\_child}
flag to identify mesh child

struct \texttt{wifi\_event\_ap\_sta\_disconnected\_t}
Argument structure for WIFI_EVENT_AP_STADISCONNECTED event

Public Members

uint8_t \texttt{mac}[6]
MAC address of the station disconnects to soft-AP

uint8_t \texttt{aid}
the aid that soft-AP gave to the station disconnects to
bool is_mesh_child
    flag to identify mesh child

uint8_t reason
    reason of disconnection

struct wifi_event_ap_probe_req_rx_t
    Argument structure for WIFI_EVENT_AP_PROBEREQRECVED event

    Public Members

    int rssi
        Received probe request signal strength

    uint8_t mac[6]
        MAC address of the station which send probe request

struct wifi_event_bss_rssi_low_t
    Argument structure for WIFI_EVENT_STA_BSS_RSSI_LOW event

    Public Members

    int32_t rssi
        RSSI value of bss

struct wifi_ftm_report_entry_t
    Argument structure for

    Public Members

    uint8_t dlog_token
        Dialog Token of the FTM frame

    int8_t rssi
        RSSI of the FTM frame received

    uint32_t rtt
        Round Trip Time in pSec with a peer

    uint64_t t1
        Time of departure of FTM frame from FTM Responder in pSec

    uint64_t t2
        Time of arrival of FTM frame at FTM Initiator in pSec
Chapter 2. API Reference

\texttt{uint64_t t3}
Time of departure of ACK from FTM Initiator in pSec

\texttt{uint64_t t4}
Time of arrival of ACK at FTM Responder in pSec

\texttt{struct wifi_event_ftm_report_t}
Argument structure for WIFI_EVENT_FTM_REPORT event

\textbf{Public Members}

\texttt{uint8_t peer_mac[6]}
MAC address of the FTM Peer

\texttt{wifi_ftm_status_t status}
Status of the FTM operation

\texttt{uint32_t rtt_raw}
Raw average Round-Trip-Time with peer in Nano-Seconds

\texttt{uint32_t rtt_est}
Estimated Round-Trip-Time with peer in Nano-Seconds

\texttt{uint32_t dist_est}
Estimated one-way distance in Centi-Meters

\texttt{wifi_ftm_report_entry_t *ftm_report_data}
Pointer to FTM Report with multiple entries, should be freed after use

\texttt{uint8_t ftm_report_num_entries}
Number of entries in the FTM Report data

\texttt{struct wifi_event_action_tx_status_t}
Argument structure for WIFI_EVENT_ACTION_TX_STATUS event

\textbf{Public Members}

\texttt{wifi_interface_t ifx}
WiFi interface to send request to

\texttt{uint32_t context}
Context to identify the request

\texttt{uint8_t da[6]}
Destination MAC address
**Chapter 2. API Reference**

```c
uint8_t status
    Status of the operation

struct wifi_event_roc_done_t
    Argument structure for WIFI_EVENT_ROC_DONE event

    Public Members

    uint32_t context
        Context to identify the request

struct wifi_event_ap_wps_rg_pin_t
    Argument structure for WIFI_EVENT_AP_WPS_RG_PIN event

    Public Members

    uint8_t pin_code[8]
        PIN code of station in enrollee mode

struct wifi_event_ap_wps_rg_fail_reason_t
    Argument structure for WIFI_EVENT_AP_WPS_RG_FAILED event

    Public Members

    wps_fail_reason_t reason
        WPS failure reason wps_fail_reason_t

    uint8_t peer_macaddr[6]
        Enrollee mac address

struct wifi_event_ap_wps_rg_success_t
    Argument structure for WIFI_EVENT_AP_WPS_RG_SUCCESS event

    Public Members

    uint8_t peer_macaddr[6]
        Enrollee mac address

struct wifi_event_nan_svc_match_t
    Argument structure for WIFIEVENT_NAN_SVC_MATCH event

    Public Members

    uint8_t subscribe_id
        Subscribe Service Identifier
```
Chapter 2. API Reference

```c
uint8_t publish_id
    Publish Service Identifier

uint8_t pub_if_mac[6]
    NAN Interface MAC of the Publisher

bool update_pub_id
    Indicates whether publisher’s service ID needs to be updated
```

**Public Members**

```c
struct wifi_event_nan_replied_t
    Argument structure for WIFI_EVENT_NAN_REPLIED event
```

**Public Members**

```c
uint8_t publish_id
    Publish Service Identifier

uint8_t subscribe_id
    Subscribe Service Identifier

uint8_t sub_if_mac[6]
    NAN Interface MAC of the Subscriber
```

**Public Members**

```c
struct wifi_event_nan_receive_t
    Argument structure for WIFI_EVENT_NAN_RECEIVE event
```

**Public Members**

```c
uint8_t inst_id
    Our Service Identifier

uint8_t peer_inst_id
    Peer’s Service Identifier

uint8_t peer_if_mac[6]
    Peer’s NAN Interface MAC

uint8_t peer_svc_info[ESP_WIFI_MAX_SVC_INFO_LEN]
    Peer Service Info
```

**Public Members**

```c
struct wifi_event_ndp_indication_t
    Argument structure for WIFI_EVENT_NDP_INDICATION event
```

**Public Members**
uint8_t publish_id
    Publish Id for NAN Service

uint8_t ndp_id
    NDP instance id

uint8_t peer_nmi[6]
    Peer’s NAN Management Interface MAC

uint8_t peer_ndi[6]
    Peer’s NAN Data Interface MAC

uint8_t svc_info[ESP_WIFI_MAX_SVC_INFO_LEN]
    Service Specific Info

struct wifi_event_ndp_confirm_t
    Argument structure for WIFI_EVENT_NDP_CONFIRM event

    Public Members

    uint8_t status
        NDP status code

    uint8_t ndp_id
        NDP instance id

    uint8_t peer_nmi[6]
        Peer’s NAN Management Interface MAC

    uint8_t peer_ndi[6]
        Peer’s NAN Data Interface MAC

    uint8_t own_ndi[6]
        Own NAN Data Interface MAC

    uint8_t svc_info[ESP_WIFI_MAX_SVC_INFO_LEN]
        Service Specific Info

struct wifi_event_ndp_terminated_t
    Argument structure for WIFI_EVENT_NDP_TERMINATED event

    Public Members

    uint8_t reason
        Termination reason code
uint8_t ndp_id
   NDP instance id

uint8_t init_ndi[6]
   Initiator’s NAN Data Interface MAC

Macros

WIFI_OFFCHAN_TX_REQ
WIFI_OFFCHAN_TX_CANCEL

WIFI_ROC_REQ
WIFI_ROC_CANCEL

WIFI_PROTOCOL_11B
WIFI_PROTOCOL_11G
WIFI_PROTOCOL_11N
WIFI_PROTOCOL_LR
WIFI_PROTOCOL_11AX

SAE_H2E_IDENTIFIER_LEN

ESP_WIFI_MAX_CONN_NUM
   max number of stations which can connect to ESP32/ESP32S3/ESP32S2 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
Chapter 2. API Reference

ESP_WIFI_NAN_MAX_SVC_SUPPORTED
ESP_WIFI_NAN_DATAPATH_MAX_PEERS
ESP_WIFI_NDP_ROLE_INITIATOR
ESP_WIFI_NDP_ROLE_RESPONDER
ESP_WIFI_MAX_SVC_NAME_LEN
ESP_WIFI_MAX_FILTER_LEN
ESP_WIFI_MAX_SVC_INFO_LEN
MAX_SSID_LEN
MAX_PASSPHRASE_LEN
MAX_WPS_AP_CRED
WIFI_STATIS_BUFFER
WIFI_STATIS_RXTX
WIFI_STATIS_HW
WIFI_STATIS_DIAG
WIFI_STATIS_PS
WIFI_STATIS_ALL

Type Definitions
typedef int (*wifi_action_rx_cb_t)(uint8_t *hdr, uint8_t *payload, size_t len, uint8_t channel)
The Rx callback function of Action Tx operations.
  
  Param hdr  pointer to the IEEE 802.11 Header structure
  Param payload  pointer to the Payload following 802.11 Header
  Param len  length of the Payload
  Param channel  channel number the frame is received on

Enumerations
enum wifi_mode_t

  Values:
enumerator WIFI_MODE_NULL
    null mode

enumerator WIFI_MODE_STA
    WiFi station mode

enumerator WIFI_MODE_AP
    WiFi soft-AP mode

enumerator WIFI_MODE_APSTA
    WiFi station + soft-AP mode

enumerator WIFI_MODE_NAN
    WiFi NAN mode

enumerator WIFI_MODE_MAX

enum wifi_interface_t
    Values:

enumerator WIFI_IF_STA
enumerator WIFI_IF_AP
enumerator WIFI_IF_NAN
enumerator WIFI_IF_MAX

enum wifi_country_policy_t
    Values:

enumerator WIFI_COUNTRY_POLICY_AUTO
    Country policy is auto, use the country info of AP to which the station is connected

enumerator WIFI_COUNTRY_POLICY_MANUAL
    Country policy is manual, always use the configured country info

enum wifi_auth_mode_t
    Values:

enumerator WIFI_AUTH_OPEN
    authenticate mode: open

enumerator WIFI_AUTH_WEP
    authenticate mode: WEP

enumerator WIFI_AUTH_WPA_PSK
    authenticate mode: WPA_PSK
enumerator WIFI_AUTH_WPA2_PSK
  authenticate mode: WPA2_PSK

enumerator WIFI_AUTH_WPA_WPA2_PSK
  authenticate mode: WPA_WPA2_PSK

enumerator WIFI_AUTH_ENTERPRISE
  authenticate mode: WiFi EAP security

enumerator WIFI_AUTH_WPA2_ENTERPRISE
  authenticate mode: WiFi EAP security

enumerator WIFI_AUTH_WPA3_PSK
  authenticate mode: WPA3_PSK

enumerator WIFI_AUTH_WPA2_WPA3_PSK
  authenticate mode: WPA2_WPA3_PSK

enumerator WIFI_AUTH_WAPI_PSK
  authenticate mode: WAPI_PSK

enumerator WIFI_AUTH_OWE
  authenticate mode: OWE

enumerator WIFI_AUTH_WPA3_ENT_192
  authenticate mode: WPA3_ENT_SUITE_B_192_BIT

enumerator WIFI_AUTH_WPA3_EXT_PSK
  authenticate mode: WPA3_PSK_EXT_KEY

enumerator WIFI_AUTH_MAX

enum wifi_err_reason_t
  Values:

enumerator WIFI_REASON_UNSPECIFIED

enumerator WIFI_REASON_AUTH_EXPIRE

enumerator WIFI_REASON_AUTH_LEAVE

enumerator WIFI_REASON_ASSOC_EXPIRE

enumerator WIFI_REASON_ASSOC_TOOMANY

enumerator WIFI_REASON_NOT_AUTHED
enumerator WIFI_REASON_NOT_ASSOCED
enumerator WIFI_REASON_ASSOC_LEAVE
enumerator WIFI_REASON_ASSOC_NOT_AUTHED
enumerator WIFI_REASON_DISASSOC_PWRCAP_BAD
enumerator WIFI_REASON_DISASSOC_SUPCHAN_BAD
enumerator WIFI_REASON_BSS_TRANSITION_DISASSOC
enumerator WIFI_REASON_IE_INVALID
enumerator WIFI_REASON_MIC_FAILURE
enumerator WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT
enumerator WIFI_REASON_GROUP_KEY_UPDATE_TIMEOUT
enumerator WIFI_REASON_IE_IN_4WAY_DIFFERS
enumerator WIFI_REASON_GROUP_CIPHER_INVALID
enumerator WIFI_REASON_PAIRWISE_CIPHER_INVALID
enumerator WIFI_REASON_AKMP_INVALID
enumerator WIFI_REASON_UNSUPP_RSN_IE_VERSION
enumerator WIFI_REASON_INVALID_RSN_IE_CAP
enumerator WIFI_REASON_802_1X_AUTH_FAILED
enumerator WIFI_REASON_CIPHER_SUITE_REJECTED
enumerator WIFI_REASON_TDLS_PEER_UNREACHABLE
enumerator WIFI_REASON_TDLS_UNSPECIFIED
enumerator WIFI_REASON_SSP_REQUESTED_DISASSOC
enumerator WIFI_REASON_NO_SSP_ROAMING_AGREEMENT
enumerator WIFI_REASON_BAD_CIPHER_OR_AKM
enumerator WIFI_REASON_NOTAUTHORIZED_THISLOCATION
enumerator WIFI_REASON_SERVICECHANGE_PERCLUDESTSS
enumerator WIFI_REASON_UNSPECIFIED_QOS
enumerator WIFI_REASON_NOTENOUGH_BANDWIDTH
enumerator WIFI_REASON_MISSING_ACKS
enumerator WIFI_REASON_EXCEEDED_TXOP
enumerator WIFI_REASON_STA_LEAVING
enumerator WIFI_REASON_END_BA
enumerator WIFI_REASON_UNKNOWN_BA
enumerator WIFI_REASON_TIMEOUT
enumerator WIFI_REASON_PEER_INITIATED
enumerator WIFI_REASON_AP_INITIATED
enumerator WIFI_REASON_INVALID_FT_ACTION_FRAME_COUNT
enumerator WIFI_REASON_INVALID_PMKID
enumerator WIFI_REASON_INVALID_MDE
enumerator WIFI_REASON_INVALID_FTE
enumerator WIFI_REASON_TRANSMISSION_LINK_ESTABLISH_FAILED
enumerator WIFI_REASON_ALTERNATIVE_CHANNEL_OCCUPIED
enumerator WIFI_REASON_BEACON_TIMEOUT
enumerator WIFI_REASON_NO_AP_FOUND
enumerator WIFI_REASON_AUTH_FAIL
enumerator WIFI_REASON_ASSOC_FAIL
enumerator WIFI_REASON_HANDSHAKE_TIMEOUT
enumerator WIFI_REASON_CONNECTION_FAIL
enumerator WIFI_REASON_AP_TSF_RESET
enumerator WIFI_REASON_ROAMING
enumerator WIFI_REASON_ASSOC_COMEBACK_TIME_TOO_LONG
enumerator WIFI_REASON_SA_QUERY_TIMEOUT
enumerator WIFI_REASON_NO_AP_FOUND_W_COMPATIBLE_SECURITY
enumerator WIFI_REASON_NO_AP_FOUND_IN_AUTHMODE_THRESHOLD
enumerator WIFI_REASON_NO_AP_FOUND_IN_RSSI_THRESHOLD

enum wifi_second_chan_t
Values:
enumerator WIFI_SECOND_CHAN_NONE
the channel width is HT20
enumerator WIFI_SECOND_CHAN_ABOVE
the channel width is HT40 and the secondary channel is above the primary channel
enumerator WIFI_SECOND_CHAN_BELOW
the channel width is HT40 and the secondary channel is below the primary channel

enum wifi_scan_type_t
Values:
enumerator WIFI_SCAN_TYPE_ACTIVE
active scan
enumerator WIFI_SCAN_TYPE_PASSIVE
passive scan

enum wifi_cipher_type_t
Values:
enumerator WIFI_CIPHER_TYPE_NONE
the cipher type is none
enumerator WIFI_CIPHER_TYPE_WEP40
the cipher type is WEP40
enumerator WIFI_CIPHER_TYPE_WEP104
the cipher type is WEP104
enumerator WIFI_CIPHER_TYPE_TKIP
the cipher type is TKIP

enumerator WIFI_CIPHER_TYPE_CCMP
the cipher type is CCMP

enumerator WIFI_CIPHER_TYPE_TKIP_CCMP
the cipher type is TKIP and CCMP

enumerator WIFI_CIPHER_TYPE_AES_CMAC128
the cipher type is AES-CMAC-128

enumerator WIFI_CIPHER_TYPE_SMS4
the cipher type is SMS4

enumerator WIFI_CIPHER_TYPE_GCMP
the cipher type is GCMP

enumerator WIFI_CIPHER_TYPE_GCMP256
the cipher type is GCMP-256

enumerator WIFI_CIPHER_TYPE_AES_GMAC128
the cipher type is AES-GMAC-128

enumerator WIFI_CIPHER_TYPE_AES_GMAC256
the cipher type is AES-GMAC-256

enumerator WIFI_CIPHER_TYPE_UNKNOWN
the cipher type is unknown

enum wifi_ant_t
WiFi antenna.
Values:

enumerator WIFI_ANT_ANT0
WiFi antenna 0

enumerator WIFI_ANT_ANT1
WiFi antenna 1

enumerator WIFI_ANT_MAX
Invalid WiFi antenna

enum wifi_scan_method_t
Values:

enumerator WIFI_FAST_SCAN
Do fast scan, scan will end after find SSID match AP
enumerator **WIFI_ALL_CHANNEL_SCAN**
   All channel scan, scan will end after scan all the channel

enum **wifi_sort_method_t**
   Values:
   enumerator **WIFI_CONNECT_AP_BY_SIGNAL**
      Sort match AP in scan list by RSSI
   enumerator **WIFI_CONNECT_AP_BY_SECURITY**
      Sort match AP in scan list by security mode

enum **wifi_ps_type_t**
   Values:
   enumerator **WIFI_PS_NONE**
      No power save
   enumerator **WIFI_PS_MIN_MODEM**
      Minimum modem power saving. In this mode, station wakes up to receive beacon every DTIM period
   enumerator **WIFI_PS_MAX_MODEM**
      Maximum modem power saving. In this mode, interval to receive beacons is determined by the listen_intent parameter in **wifi_sta_config_t**

enum **wifi_bandwidth_t**
   Values:
   enumerator **WIFI_BW_HT20**
   enumerator **WIFI_BW_HT40**

enum **wifi_sae_pwe_method_t**
   Configuration for SAE PWE derivation
   Values:
   enumerator **WPA3_SAE_PWE_UNSPECIFIED**
   enumerator **WPA3_SAE_PWE_HUNT_AND_PECK**
   enumerator **WPA3_SAE_PWE_HASH_TO_ELEMENT**
   enumerator **WPA3_SAE_PWE_BOTH**

enum **wifi_sae_pk_mode_t**
   Configuration for SAE-PK
   Values:
enumerator WPA3_SAE_PK_MODE_AUTOMATIC

enumerator WPA3_SAE_PK_MODE_ONLY

enumerator WPA3_SAE_PK_MODE_DISABLED

div

enum wifi_storage_t

Values:

enumerator WIFI_STORAGE_FLASH
    all configuration will store in both memory and flash

enumerator WIFI_STORAGE_RAM
    all configuration will only store in the memory

div

enum wifi_vendor_ie_type_t

Vendor Information Element type.
Determines the frame type that the IE will be associated with.

Values:

enumerator WIFI_VND_IE_TYPE_BEACON

enumerator WIFI_VND_IE_TYPE_PROBE_REQ

enumerator WIFI_VND_IE_TYPE_PROBE_RESP

enumerator WIFI_VND_IE_TYPE_ASSOC_REQ

enumerator WIFI_VND_IE_TYPE_ASSOC_RESP

div

enum wifi_vendor_ie_id_t

Vendor Information Element index.
Each IE type can have up to two associated vendor ID elements.

Values:

enumerator WIFI_VND_IE_ID_0

enumerator WIFI_VND_IE_ID_1

div

enum wifi_phy_mode_t

Operation Phymode.

Values:

enumerator WIFI_PHY_MODE_LR
    PHY mode for Low Rate
enumerator WIFI_PHY_MODE_11B
    PHY mode for 11b
enumerator WIFI_PHY_MODE_11G
    PHY mode for 11g
enumerator WIFI_PHY_MODE_HT20
    PHY mode for Bandwidth HT20
enumerator WIFI_PHY_MODE_HT40
    PHY mode for Bandwidth HT40
enumerator WIFI_PHY_MODE_HE20
    PHY mode for Bandwidth HE20

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:

text
    enumerator WIFI_PKT_MGMT
        Management frame, indicates ‘buf’ argument is wifi_promiscuous_pkt_t

text
    enumerator WIFI_PKT_CTRL
        Control frame, indicates ‘buf’ argument is wifi_promiscuous_pkt_t

text
    enumerator WIFI_PKT_DATA
        Data frame, indicates ‘buf’ argument is wifi_promiscuous_pkt_t

text
    enumerator 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:

text
    enumerator WIFI_ANT_MODE_ANT0
        Enable WiFi antenna 0 only

text
    enumerator WIFI_ANT_MODE_ANT1
        Enable WiFi antenna 1 only

text
    enumerator WIFI_ANT_MODE_AUTO
        Enable WiFi antenna 0 and 1, automatically select an antenna

text
    enumerator WIFI_ANT_MODE_MAX
        Invalid WiFi enabled antenna
enum `wifi_nan_service_type_t`

NAN Services types.

Values:

enumerator `NAN_PUBLISH_SOLICITED`
Send unicast Publish frame to Subscribers that match the requirement

enumerator `NAN_PUBLISH_UNSOLICITED`
Send broadcast Publish frames in every Discovery Window(DW)

enumerator `NAN_SUBSCRIBE_ACTIVE`
Send broadcast Subscribe frames in every DW

enumerator `NAN_SUBSCRIBE_PASSIVE`
Passively listens to Publish frames

enum `wifi_phy_rate_t`

WiFi PHY rate encodings.

Values:

enumerator `WIFI_PHY_RATE_1M_L`
1 Mbps with long preamble

enumerator `WIFI_PHY_RATE_2M_L`
2 Mbps with long preamble

enumerator `WIFI_PHY_RATE_5M_L`
5.5 Mbps with long preamble

enumerator `WIFI_PHY_RATE_11M_L`
11 Mbps with long preamble

enumerator `WIFI_PHY_RATE_2M_S`
2 Mbps with short preamble

enumerator `WIFI_PHY_RATE_5M_S`
5.5 Mbps with short preamble

enumerator `WIFI_PHY_RATE_11M_S`
11 Mbps with short preamble

enumerator `WIFI_PHY_RATE_48M`
48 Mbps

enumerator `WIFI_PHY_RATE_24M`
24 Mbps
enumerator WIFI_PHY_RATE_12M
   12 Mbps

enumerator WIFI_PHY_RATE_6M
   6 Mbps

enumerator WIFI_PHY_RATE_54M
   54 Mbps

enumerator WIFI_PHY_RATE_36M
   36 Mbps

enumerator WIFI_PHY_RATE_18M
   18 Mbps

enumerator WIFI_PHY_RATE_9M
   9 Mbps
   rate table and guard interval information for each MCS rate

enumerator WIFI_PHY_RATE_MCS0_LGI
   MCS0 with long GI

enumerator WIFI_PHY_RATE_MCS1_LGI
   MCS1 with long GI

enumerator WIFI_PHY_RATE_MCS2_LGI
   MCS2 with long GI

enumerator WIFI_PHY_RATE_MCS3_LGI
   MCS3 with long GI

enumerator WIFI_PHY_RATE_MCS4_LGI
   MCS4 with long GI

enumerator WIFI_PHY_RATE_MCS5_LGI
   MCS5 with long GI

enumerator WIFI_PHY_RATE_MCS6_LGI
   MCS6 with long GI

enumerator WIFI_PHY_RATE_MCS7_LGI
   MCS7 with long GI

enumerator WIFI_PHY_RATE_MCS0_SGI
   MCS0 with short GI

enumerator WIFI_PHY_RATE_MCS1_SGI
   MCS1 with short GI
enumerator WIFI_PHY_RATE_MCS2_SGI
    MCS2 with short GI
enumerator WIFI_PHY_RATE_MCS3_SGI
    MCS3 with short GI
enumerator WIFI_PHY_RATE_MCS4_SGI
    MCS4 with short GI
enumerator WIFI_PHY_RATE_MCS5_SGI
    MCS5 with short GI
enumerator WIFI_PHY_RATE_MCS6_SGI
    MCS6 with short GI
enumerator WIFI_PHY_RATE_MCS7_SGI
    MCS7 with short GI
enumerator WIFI_PHY_RATE_LORA_250K
    250 Kbps
enumerator WIFI_PHY_RATE_LORA_500K
    500 Kbps
enumerator WIFI_PHY_RATE_MAX

enum wifi_event_t
    WiFi event declarations
    Values:
enumerator WIFI_EVENT_WIFI_READY
    WiFi ready
enumerator WIFI_EVENT_SCAN_DONE
    Finished scanning AP
enumerator WIFI_EVENT_STA_START
    Station start
enumerator WIFI_EVENT_STA_STOP
    Station stop
enumerator WIFI_EVENT_STA_CONNECTED
    Station connected to AP
enumerator WIFI_EVENT_STA_DISCONNECTED
    Station disconnected from AP
enumerator `WIFI_EVENT_STA_AUTHMODE_CHANGE`
   the auth mode of AP connected by device’s station changed

enumerator `WIFI_EVENT_STA_WPS_ER_SUCCESS`
   Station wps succeeds in enrollee mode

enumerator `WIFI_EVENT_STA_WPS_ER_FAILED`
   Station wps fails in enrollee mode

enumerator `WIFI_EVENT_STA_WPS_ER_TIMEOUT`
   Station wps timeout in enrollee mode

enumerator `WIFI_EVENT_STA_WPS_ER_PIN`
   Station wps pin code in enrollee mode

enumerator `WIFI_EVENT_STA_WPS_ER_PBC_OVERLAP`
   Station wps overlap in enrollee mode

enumerator `WIFI_EVENT_AP_START`
   Soft-AP start

enumerator `WIFI_EVENT_AP_STOP`
   Soft-AP stop

enumerator `WIFI_EVENT_AP_STACONNECTED`
   a station connected to Soft-AP

enumerator `WIFI_EVENT_AP_STADISCONNECTED`
   a station disconnected from Soft-AP

enumerator `WIFI_EVENT_AP_PROBEREQRECEIVED`
   Receive probe request packet in soft-AP interface

enumerator `WIFI_EVENT_FTM_REPORT`
   Receive report of FTM procedure

enumerator `WIFI_EVENT_STA_BSS_RSSI_LOW`
   AP’s RSSI crossed configured threshold

enumerator `WIFI_EVENT_ACTION_TX_STATUS`
   Status indication of Action Tx operation

enumerator `WIFI_EVENT_ROC_DONE`
   Remain-on-Channel operation complete

enumerator `WIFI_EVENT_STA_BEACON_TIMEOUT`
   Station beacon timeout
enumerator WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START
  Connectionless module wake interval start
enumerator WIFI_EVENT_AP_WPS_RG_SUCCESS
  Soft-AP wps succeeds in registrar mode
enumerator WIFI_EVENT_AP_WPS_RG_FAILED
  Soft-AP wps fails in registrar mode
enumerator WIFI_EVENT_AP_WPS_RG_TIMEOUT
  Soft-AP wps timeout in registrar mode
enumerator WIFI_EVENT_AP_WPS_RG_PIN
  Soft-AP wps pin code in registrar mode
enumerator WIFI_EVENT_AP_WPS_RG_PBC_OVERLAP
  Soft-AP wps overlap in registrar mode
enumerator WIFI_EVENT_ITWT_SETUP
  iTWT setup
enumerator WIFI_EVENT_ITWT_TEARDOWN
  iTWT teardown
enumerator WIFI_EVENT_ITWT_PROBE
  iTWT probe
enumerator WIFI_EVENT_ITWT_SUSPEND
  iTWT suspend
enumerator WIFI_EVENT_NAN_STARTED
  NAN Discovery has started
enumerator WIFI_EVENT_NAN_STOPPED
  NAN Discovery has stopped
enumerator WIFI_EVENT_NAN_SVC_MATCH
  NAN Service Discovery match found
enumerator WIFI_EVENT_NAN_REPLIED
  Replied to a NAN peer with Service Discovery match
enumerator WIFI_EVENT_NAN_RECEIVE
  Received a Follow-up message
enumerator WIFI_EVENT_NDP_INDICATION
  Received NDP Request from a NAN Peer
enumerator WIFI_EVENT_NDP_CONFIRM
    NDP Confirm Indication

enumerator WIFI_EVENT_NDP_TERMINATED
    NAN Datapath terminated indication

enumerator 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:

    enumerator WPS_FAIL_REASON_NORMAL
        WPS normal fail reason

    enumerator WPS_FAIL_REASON_RECV_M2D
        WPS receive M2D frame

    enumerator WPS_FAIL_REASON_MAX

enum wifi_ftm_status_t
    FTM operation status types.
    Values:

    enumerator FTM_STATUS_SUCCESS
        FTM exchange is successful

    enumerator FTM_STATUS_UNSUPPORTED
        Peer does not support FTM

    enumerator FTM_STATUS_CONF_REJECTED
        Peer rejected FTM configuration in FTM Request

    enumerator FTM_STATUS_NO_RESPONSE
        Peer did not respond to FTM Requests

    enumerator FTM_STATUS_FAIL
        Unknown error during FTM exchange

enum wps_fail_reason_t
    Values:

    enumerator WPS_AP_FAIL_REASON_NORMAL
        WPS normal fail reason

    enumerator WPS_AP_FAIL_REASON_CONFIG
        WPS failed due to incorrect config
enumerator **WPS_AP_FAIL_REASON_AUTH**

WPS failed during auth

enumerator **WPS_AP_FAIL_REASON_MAX**

**Header File**

- components/wpa_supplicant/esp_supplicant/include/esp_eap_client.h
- This header file can be included with:
  
  ```c
  #include "esp_eap_client.h"
  ```

- This header file is a part of the API provided by the `wpa_supplicant` component. To declare that your component depends on `wpa_supplicant`, add the following to your CMakeLists.txt:
  
  ```cmake
  REQUIRES wpa_supplicant
  or
  PRIV_REQUIRES wpa_supplicant
  ```

**Functions**

**esp_err_t esp_wifi_staEnterpriseEnable**(void)

Enable EAP authentication (WiFi Enterprise) for the station mode.

This function enables Extensible Authentication Protocol (EAP) authentication for the Wi-Fi station mode. When EAP authentication is enabled, the ESP device will attempt to authenticate with the configured EAP credentials when connecting to a secure Wi-Fi network.

**Note:** Before calling this function, ensure that the Wi-Fi configuration and EAP credentials (such as username and password) have been properly set using the appropriate configuration APIs.

**Returns**

- ESP_OK: EAP authentication enabled successfully.
- ESP_ERR_NO_MEM: Failed to enable EAP authentication due to memory allocation failure.

**esp_err_t esp_wifi_staEnterpriseDisable**(void)

Disable EAP authentication (WiFi Enterprise) for the station mode.

This function disables Extensible Authentication Protocol (EAP) authentication for the Wi-Fi station mode. When EAP authentication is disabled, the ESP device will not attempt to authenticate using EAP credentials when connecting to a secure Wi-Fi network.

**Note:** Disabling EAP authentication may cause the device to connect to the Wi-Fi network using other available authentication methods, if configured using `esp_wifi_set_config()`.

**Returns**

- ESP_OK: EAP authentication disabled successfully.
- ESP_ERR_INVALID_STATE: EAP client is in an invalid state for disabling.

**esp_err_t esp_eap_client_set_identity**(const unsigned char *identity, int len)

Set identity for PEAP/TTLS authentication method.

This function sets the identity to be used during PEAP/TTLS authentication.
Parameters

• `identity` - [in] Pointer to the identity data.
• `len` - [in] Length of the identity data (limited to 1~127 bytes).

Returns

• ESP_OK: The identity was set successfully.
• ESP_ERR_INVALID_ARG: Invalid argument (len <= 0 or len >= 128).
• ESP_ERR_NO_MEM: Memory allocation failure.

void `esp_eap_client_clear_identity` (void)
Clear the previously set identity for PEAP/TTLS authentication.

This function clears the identity that was previously set for the EAP client. After calling this function, the EAP client will no longer use the previously configured identity during the authentication process.

esp_err_t `esp_eap_client_set_username` (const unsigned char *username, int len)
Set username for PEAP/TTLS authentication method.

This function sets the username to be used during PEAP/TTLS authentication.

Parameters

• `username` - [in] Pointer to the username data.
• `len` - [in] Length of the username data (limited to 1~127 bytes).

Returns

• ESP_OK: The username was set successfully.
• ESP_ERR_INVALID_ARG: Failed due to an invalid argument (len <= 0 or len >= 128).
• ESP_ERR_NO_MEM: Failed due to memory allocation failure.

void `esp_eap_client_clear_username` (void)
Clear username for PEAP/TTLS method.

This function clears the previously set username for the EAP client.

esp_err_t `esp_eap_client_set_password` (const unsigned char *password, int len)
Set password for PEAP/TTLS authentication method.

This function sets the password to be used during PEAP/TTLS authentication.

Parameters

• `password` - [in] Pointer to the password data.
• `len` - [in] Length of the password data (len > 0).

Returns

• ESP_OK: The password was set successfully.
• ESP_ERR_INVALID_ARG: Failed due to an invalid argument (len <= 0).
• ESP_ERR_NO_MEM: Failed due to memory allocation failure.

void `esp_eap_client_clear_password` (void)
Clear password for PEAP/TTLS method.

This function clears the previously set password for the EAP client.

esp_err_t `esp_eap_client_set_new_password` (const unsigned char *new_password, int len)
Set a new password for MSCHAPv2 authentication method.

This function sets the new password to be used during MSCHAPv2 authentication. The new password is used to substitute the old password when an eap-mschapv2 failure request message with error code ERROR_PASSWD_EXPIRED is received.

Parameters

• `new_password` - [in] Pointer to the new password data.
• `len` - [in] Length of the new password data.

Returns

• ESP_OK: The new password was set successfully.
• ESP_ERR_INVALID_ARG: Failed due to an invalid argument (len <= 0).
• ESP_ERR_NO_MEM: Failed due to memory allocation failure.
void esp_eap_client_clear_new_password (void)
Clear new password for MSCHAPv2 method.
This function clears the previously set new password for the EAP client.

esp_err_t esp_eap_client_set_ca_cert (const unsigned char *ca_cert, int ca_cert_len)
Set CA certificate for EAP authentication.
This function sets the Certificate Authority (CA) certificate to be used during EAP authentication. The CA certificate is passed to the EAP client module through a global pointer.

Parameters
• ca_cert – [in] Pointer to the CA certificate data.
• ca_cert_len – [in] Length of the CA certificate data.

Returns
• ESP_OK: The CA certificate was set successfully.

void esp_eap_client_clear_ca_cert (void)
Clear the previously set Certificate Authority (CA) certificate for EAP authentication.
This function clears the CA certificate that was previously set for the EAP client. After calling this function, the EAP client will no longer use the previously configured CA certificate during the authentication process.

esp_err_t esp_eap_client_set_certificate_and_key (const unsigned char *client_cert, int client_cert_len, const unsigned char *private_key, int private_key_len, const unsigned char *private_key_password, int private_key_passwd_len)
Set client certificate and private key for EAP authentication.
This function sets the client certificate and private key to be used during authentication. Optionally, a private key password can be provided for encrypted private keys.

Attention 1. The client certificate, private key, and private key password are provided as pointers to the respective data arrays.
Attention 2. The client_cert, private_key, and private_key_password should be zero-terminated.

Parameters
• client_cert – [in] Pointer to the client certificate data.
• client_cert_len – [in] Length of the client certificate data.
• private_key – [in] Pointer to the private key data.
• private_key_len – [in] Length of the private key data (limited to 1~4096 bytes).
• private_key_password – [in] Pointer to the private key password data (optional).
• private_key_passwd_len – [in] Length of the private key password data (can be 0 for no password).

Returns
• ESP_OK: The certificate, private key, and password (if provided) were set successfully.

void esp_eap_client_clear_certificate_and_key (void)
Clear the previously set client certificate and private key for EAP authentication.
This function clears the client certificate and private key that were previously set for the EAP client. After calling this function, the EAP client will no longer use the previously configured certificate and private key during the authentication process.

esp_err_t esp_eap_client_set_disable_time_check (bool disable)
Set EAP client certificates time check (disable or not).
This function enables or disables the time check for EAP client certificates. When disabled, the certificates’ expiration time will not be checked during the authentication process.

Parameters disable – [in] True to disable EAP client certificates time check, false to enable it.
Returns

• ESP_OK: The EAP client certificates time check setting was updated successfully.

`esp_err_t esp_eap_client_get_disable_time_check (bool *disable)`

Get EAP client certificates time check status.

This function retrieves the current status of the EAP client certificates time check.

Parameters disable -[out] Pointer to a boolean variable to store the disable status.

Returns

• ESP_OK: The status of EAP client certificates time check was retrieved successfully.

`esp_err_t esp_eap_client_set_ttls_phase2_method (esp_eap_ttls_phase2_types type)`

Set EAP-TTLS phase 2 method.

This function sets the phase 2 method to be used during EAP-TTLS authentication.

Parameters type –[in] The type of phase 2 method to be used (e.g., EAP, MSCHAPv2, MSCHAP, PAP, CHAP).

Returns

• ESP_OK: The EAP-TTLS phase 2 method was set successfully.

`esp_err_t esp_eap_client_set_suiteb_192bit_certification (bool enable)`

Enable or disable Suite-B 192-bit certification checks.

This function enables or disables the 192-bit Suite-B certification checks during EAP-TLS authentication. Suite-B is a set of cryptographic algorithms which generally are considered more secure.

Parameters enable –[in] True to enable 192-bit Suite-B certification checks, false to disable it.

Returns

• ESP_OK: The 192-bit Suite-B certification checks were set successfully.

`esp_err_t esp_eap_client_set_pac_file (const unsigned char *pac_file, int pac_file_len)`

Set the PAC (Protected Access Credential) file for EAP-FAST authentication.

EAP-FAST requires a PAC file that contains the client’s credentials.

Attention 1. For files read from the file system, length has to be decremented by 1 byte.
Attention 2. Disabling the ESP_WIFI_MBEDTLS_TLS_CLIENT config is required to use EAP-FAST.

Parameters

• pac_file –[in] Pointer to the PAC file buffer.
• pac_file_len –[in] Length of the PAC file buffer.

Returns

• ESP_OK: The PAC file for EAP-FAST authentication was set successfully.

`esp_err_t esp_eap_client_set_fast_params (esp_eap_fast_config config)`

Set the parameters for EAP-FAST Phase 1 authentication.

EAP-FAST supports Fast Provisioning, where clients can be authenticated faster using precomputed keys (PAC). This function allows configuring parameters for Fast Provisioning.

Attention 1. Disabling the ESP_WIFI_MBEDTLS_TLS_CLIENT config is required to use EAP-FAST.

Parameters config –[in] Configuration structure with Fast Provisioning parameters.

Returns

• ESP_OK: The parameters for EAP-FAST Phase 1 authentication were set successfully.
**esp_err_t esp_eap_client_use_default_cert_bundle (bool use_default_bundle)**

Use the default certificate bundle for EAP authentication.

By default, the EAP client uses a built-in certificate bundle for server verification. Enabling this option allows the use of the default certificate bundle.

**Parameters**

- **use_default_bundle** [in] True to use the default certificate bundle, false to use a custom bundle.

**Returns**

- ESP_OK: The option to use the default certificate bundle was set successfully.

---

**Structures**

**struct esp_eap_fast_config**

Configuration settings for EAP-FAST (Extensible Authentication Protocol - Flexible Authentication via Secure Tunneling).

This structure defines the configuration options that can be used to customize the behavior of the EAP-FAST authentication protocol, specifically for Fast Provisioning and PAC (Protected Access Credential) handling.

**Public Members**

- **int fast_provisioning**
  
  Enable or disable Fast Provisioning in EAP-FAST (0 = disabled, 1 = enabled)

- **int fast_max_pac_list_len**
  
  Maximum length of the PAC (Protected Access Credential) list

- **bool fast_pac_format_binary**
  
  Set to true for binary format PAC, false for ASCII format PAC

**Enumerations**

**enum esp_eap_ttls_phase2_types**

Enumeration of phase 2 authentication types for EAP-TTLS.

This enumeration defines the supported phase 2 authentication methods that can be used in the EAP-TTLS (Extensible Authentication Protocol - Tunneled Transport Layer Security) protocol for the second authentication phase.

**Values:**

- **enumerator ESP_EAP_TTLS_PHASE2_EAP**
  
  EAP (Extensible Authentication Protocol)

- **enumerator ESP_EAP_TTLS_PHASE2_MSCHAPV2**
  
  MS-CHAPv2 (Microsoft Challenge Handshake Authentication Protocol - Version 2)

- **enumerator ESP_EAP_TTLS_PHASE2_MSCHAP**
  
  MS-CHAP (Microsoft Challenge Handshake Authentication Protocol)

- **enumerator ESP_EAP_TTLS_PHASE2_PAP**
  
  PAP (Password Authentication Protocol)
Chapter 2. API Reference

enumerator **ESP_EAP_TTLS_PHASE2_CHAP**

CHAP (Challenge Handshake Authentication Protocol)

**Header File**

- components/wpa_supplicant/esp_supplicant/include/esp_wps.h
- This header file can be included with:

```
#include "esp_wps.h"
```

- This header file is a part of the API provided by the wpa_supplicant component. To declare that your component depends on wpa_supplicant, add the following to your CMakeLists.txt:

```
REQUIRES wpa_supplicant
```

or

```
PRIV_REQUIRES wpa_supplicant
```

**Functions**

```
esp_err_t esp_wifi_wps_enable (const esp_wps_config_t *config)
```

Enable Wi-Fi WPS function.

**Parameters**

- `config` : WPS config to be used in connection

**Returns**

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_FAIL : wps initialization fails

```
esp_err_t esp_wifi_wps_disable (void)
```

Disable Wi-Fi WPS function and release resource it taken.

**Returns**

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on

```
esp_err_t esp_wifi_wps_start (int timeout_ms)
```

Start WPS session.

**Attention**  WPS can only be used when station is enabled. WPS needs to be enabled first for using this API.

**Parameters**

- `timeout_ms` : deprecated: This argument’s value will have not effect in functionality of API. The argument will be removed in future. The app should start WPS and register for WIFI events to get the status. WPS status is updated through WPS events. See wifi_event_t enum for more info.

**Returns**

- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_ERR_WIFI_WPS_SM : wps state machine is not initialized
- ESP_FAIL : wps initialization fails

```
esp_err_t esp_wifi_ap_wps_enable (const esp_wps_config_t *config)
```

Enable Wi-Fi AP WPS function.

**Attention**  WPS can only be used when softAP is enabled.
**Parameters** `config`—wps configuration to be used.

**Returns**
- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_FAIL : wps initialization fails

```c
esp_err_t esp_wifi_ap_wps_disable (void)
```

Disable Wi-Fi SoftAP WPS function and release resource it taken.

**Returns**
- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on

```c
esp_err_t esp_wifi_ap_wps_start (const unsigned char* pin)
```

WPS starts to work.

**Attention** WPS can only be used when softAP is enabled.

**Parameters** `pin`—Pin to be used in case of WPS mode is pin. If Pin is not provided, device will use the pin generated/provided during `esp_wifi_ap_wps_enable()` and reported in WIFI_EVENT_AP_WPS_RG_PIN

**Returns**
- ESP_OK : succeed
- ESP_ERR_WIFI_WPS_TYPE : wps type is invalid
- ESP_ERR_WIFI_WPS_MODE : wifi is not in station mode or sniffer mode is on
- ESP_ERR_WIFI_WPS_SM : wps state machine is not initialized
- ESP_FAIL : wps initialization fails

**Structures**

```c
struct wps_factory_information_t
```

Structure representing WPS factory information for ESP device.

This structure holds various strings representing factory information for a device, such as the manufacturer, model number, model name, and device name. Each string is a null-terminated character array. If any of the strings are empty, the default values are used.

**Public Members**

```c
char manufacturer[WPS_MAX_MANUFACTURER_LEN]
```

Manufacturer of the device. If empty, the default manufacturer is used.

```c
char model_number[WPS_MAX_MODEL_NUMBER_LEN]
```

Model number of the device. If empty, the default model number is used.

```c
char model_name[WPS_MAX_MODEL_NAME_LEN]
```

Model name of the device. If empty, the default model name is used.

```c
char device_name[WPS_MAX_DEVICE_NAME_LEN]
```

Device name. If empty, the default device name is used.
struct esp_wps_config_t

Structure representing configuration settings for WPS (Wi-Fi Protected Setup).

This structure encapsulates various configuration settings for WPS, including the WPS type (PBC or PIN),
factory information that will be shown in the WPS Information Element (IE), and a PIN if the WPS type is set
to PIN.

Public Members

wps_type_t wps_type

The type of WPS to be used (PBC or PIN).

wps_factory_information_t factory_info

Factory information to be shown in the WPS Information Element (IE). Vendor can choose to display
their own information.

char pin[PIN_LEN]

WPS PIN (Personal Identification Number) used when wps_type is set to WPS_TYPE_PIN.

Macros

ESP_ERR_WIFI_REGISTRAR

WPS registrar is not supported

ESP_ERR_WIFI_WPS_TYPE

WPS type error

ESP_ERR_WIFI_WPS_SM

WPS state machine is not initialized

WPS_MAX_MANUFACTURER_LEN

Maximum length of the manufacturer name in WPS information

WPS_MAX_MODEL_NUMBER_LEN

Maximum length of the model number in WPS information

WPS_MAX_MODEL_NAME_LEN

Maximum length of the model name in WPS information

WPS_MAX_DEVICE_NAME_LEN

Maximum length of the device name in WPS information

PIN_LEN

The length of the WPS PIN (Personal Identification Number).

WPS_CONFIG_INIT_DEFAULT (type)

Initialize a default WPS configuration structure with specified WPS type.

This macro initializes a esp_wps_config_t structure with default values for the specified WPS type. It
sets the WPS type, factory information (including default manufacturer, model number, model name, and
device name), and a default PIN value if applicable.
Parameters
  • **type** - The WPS type to be used (PBC or PIN).

Returns An initialized `esp_wps_config_t` structure with the specified WPS type and default values.

Type Definitions

typedef enum **wps_type**

Enumeration of WPS (Wi-Fi Protected Setup) types.

Enumerations

enum **wps_type**

Enumeration of WPS (Wi-Fi Protected Setup) types.

Values:

enumerator **WPS_TYPE_DISABLE**

  WPS is disabled

enumerator **WPS_TYPE_PBC**

  WPS Push Button Configuration method

enumerator **WPS_TYPE_PIN**

  WPS PIN (Personal Identification Number) method

enumerator **WPS_TYPE_MAX**

  Maximum value for WPS type enumeration

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_rrm.h
  - This header file can be included with:

```c
#include "esp_rrm.h"
```

- This header file is a part of the API provided by the `wpa_supplicant` component. To declare that your component depends on `wpa_supplicant`, add the following to your CMakeLists.txt:

```cmake
REQUIRES wpa_supplicant
```

or

```cmake
PRIV_REQUIRES wpa_supplicant
```

Functions

```c
int esp_rrm_send_neighbor_rep_request (neighbor_rep_request_cb cb, void *cb_ctx)
```

Send Radio measurement neighbor report request to connected AP.

Parameters
  • **cb** - callback function for neighbor report
  • **cb_ctx** - callback context

Returns
  • 0: success
  • -1: AP does not support RRM
  • -2: station not connected to AP
bool esp_rrm_is_rrm_supported_connection(void)
Check RRM capability of connected AP.

Returns
• true: AP supports RRM
• false: AP does not support RRM or station not connected to AP

Type Definitions
typedef void (*neighbor_rep_request_cb)(void *ctx, const uint8_t *report, size_t report_len)
Callback function type to get neighbor report.

Param ctx
neighbor report context
Param report
eighbor report
Param report_len
neighbor report length
Returns
• void

Header File
• components/wpa_supplicant/esp_supplicant/include/esp_wnm.h
• This header file can be included with:

```c
#include "esp_wnm.h"
```
• This header file is a part of the API provided by the wpa_supplicant component. To declare that your component depends on wpa_supplicant, add the following to your CMakeLists.txt:

```cmake
REQUIRES wpa_supplicant
```

or

```cmake
PRIV_REQUIRES wpa_supplicant
```

Functions
int esp_wnm_send_bss_transition_mgmt_query(enum btm_query_reason query_reason, const char *btm_candidates, int cand_list)
Send bss transition query to connected AP.

Parameters
• query_reason – reason for sending query
• btm_candidates – btm candidates list if available
• cand_list – whether candidate list to be included from scan results available in supplicant’s cache.

Returns
• 0: success
• -1: AP does not support BTM
• -2: station not connected to AP

bool esp_wnm_is_btm_supported_connection(void)
Check bss transition capability of connected AP.

Returns
• true: AP supports BTM
• false: AP does not support BTM or station not connected to AP

Enumerations
enum **btm_query_reason**

enum btm_query_reason: Reason code for sending btm query

*Values:*

enumerator **REASON_UNSPECIFIED**

enumerator **REASON_FRAME_LOSS**

enumerator **REASON_DELAY**

enumerator **REASON_BANDWIDTH**

enumerator **REASON_LOAD_BALANCE**

enumerator **REASON_RSSI**

enumerator **REASON_RETRANSMISSIONS**

enumerator **REASON_INTERFERENCE**

enumerator **REASON_GRAY_ZONE**

enumerator **REASON_PREMIUM_AP**

**Header File**

- components/wpa_supplicant/esp_supplicant/include/esp_mbo.h
- This header file can be included with:

  ```
  #include "esp_mbo.h"
  ```

- This header file is a part of the API provided by the wpa_supplicant component. To declare that your component depends on wpa_supplicant, add the following to your CMakeLists.txt:

  ```
  REQUIRE wpa_supplicant
  ```
  or
  ```
  PRIV_REQUIRE wpa_supplicant
  ```

**Functions**

```
int esp_mbo_update_non_pref_chan (struct non_pref_chan_s **non_pref_chan)
```

Update channel preference for MBO IE.

*Parameters non_pref_chan* - Non preference channel list

*Returns*

- 0: success else failure
## Structures

### `struct non_pref_chan`

Structure representing a non-preferred channel in a wireless network. This structure encapsulates information about a non-preferred channel including the reason for its non-preference, the operating class, channel number, and preference level.

#### Public Members

- `enum non_pref_chan_reason reason`
  - Reason for the channel being non-preferred
- `uint8_t oper_class`
  - Operating class of the channel
- `uint8_t chan`
  - Channel number
- `uint8_t preference`
  - Preference level of the channel

### `struct non_pref_chan_s`

Structure representing a list of non-preferred channels in a wireless network. This structure encapsulates information about a list of non-preferred channels including the number of non-preferred channels and an array of structures representing individual non-preferred channels.

#### Public Members

- `size_t non_pref_chan_num`
  - Number of non-preferred channels in the list
- `struct non_pref_chan chan[]`
  - Array of structures representing individual non-preferred channels

## Enumerations

### `enum non_pref_chan_reason`

Enumeration of reasons for a channel being non-preferred in a wireless network. This enumeration defines various reasons why a specific channel might be considered non-preferred in a wireless network configuration.

#### Values:

- `NON_PREF_CHAN_REASON_UNSPECIFIED`
  - Unspecified reason for non-preference
- `NON_PREF_CHAN_REASON_RSSI`
  - Non-preferred due to low RSSI (Received Signal Strength Indication)
Wi-Fi Easy Connect™ (DPP)

Wi-Fi Easy Connect™, also known as Device Provisioning Protocol (DPP) or Easy Connect, is a provisioning protocol certified by Wi-Fi Alliance. It is a secure and standardized provisioning protocol for configuration of Wi-Fi Devices. With Easy Connect, adding a new device to a network is as simple as scanning a QR Code. This reduces complexity and enhances user experience while onboarding devices without UI like Smart Home and IoT products. Unlike old protocols like Wi-Fi Protected Setup (WPS), Wi-Fi Easy Connect in corporates strong encryption through public key cryptography to ensure networks remain secure as new devices are added.

Easy Connect brings many benefits in the user experience:

- Simple and intuitive to use; no lengthy instructions to follow for new device setup
- No need to remember and enter passwords into the device being provisioned
- Works with electronic or printed QR codes, or human-readable strings
- Supports both WPA2 and WPA3 networks

Please refer to Wi-Fi Alliance’s official page on Easy Connect for more information.

ESP32-S2 supports Enrollee mode of Easy Connect with QR Code as the provisioning method. A display is required to display this QR Code. Users can scan this QR Code using their capable device and provision the ESP32-S2 to their Wi-Fi network. The provisioning device needs to be connected to the AP which need not support Wi-Fi Easy Connect™.

Easy Connect is still an evolving protocol. Of known platforms that support the QR Code method are some Android smartphones with Android 10 or higher. To use Easy Connect, no additional App needs to be installed on the supported smartphone.

Application Example Example on how to provision ESP32-S2 using a supported smartphone: wifi/wifi_easy_connect/dpp-enrollee.

API Reference

Header File

- components/wpa_supplicant/esp_supplicant/include/esp_dpp.h
- This header file can be included with:

```c
#include "esp_dpp.h"
```

- This header file is a part of the API provided by the wpa_supplicant component. To declare that your component depends on wpa_supplicant, add the following to your CMakeLists.txt:

```cmake
REQUIRES wpa_supplicant
```

or

```cmake
PRIV_REQUIRES wpa_supplicant
```
## Functions

### esp_err_t esp_supp_dpp_init(esp_supp_dpp_event_cb_t evt_cb)

Initialize DPP Supplicant.

- **Return**
  - ESP_OK: Success
  - ESP_FAIL: Failure

- **Parameters**
  - `evt_cb` - Callback function to receive DPP related events

### esp_supp_dpp_deinit(void)

De-initialize DPP Supplicant.

- **Frees memory from DPP Supplicant Data Structures.**

### esp_err_t esp_supp_dpp_bootstrap_gen(const char* chan_list, esp_supp_dpp_bootstrap_t type, const char* key, const char* info)

Generates Bootstrap Information as an Enrollee.

- **Parameters**
  - `chan_list` - List of channels device will be available on for listening
  - `type` - Bootstrap method type, only QR Code method is supported for now.
  - `key` - (Optional) 32 byte Raw Private Key for generating a Bootstrapping Public Key
  - `info` - (Optional) Ancilliary Device Information like Serial Number

- **Returns**
  - ESP_OK: Success
  - ESP_FAIL: Failure

### esp_err_t esp_supp_dpp_start_listen(void)

Start listening on Channels provided during esp_supp_dpp_bootstrap_gen.

- **Listens on every Channel from Channel List for a pre-defined wait time.**

- **Returns**
  - ESP_OK: Success
  - ESP_FAIL: Generic Failure
  - ESP_ERR_INVALID_STATE: ROC attempted before WiFi is started
  - ESP_ERR_NO_MEM: Memory allocation failed while posting ROC request

### void esp_supp_dpp_stop_listen(void)

Stop listening on Channels.
Chapter 2. API Reference

Stops listening on Channels and cancels ongoing listen operation.

Macros

**ESP_ERR_DPP_FAILURE**
Generic failure during DPP Operation

**ESP_ERR_DPP_TX_FAILURE**
DPP Frame Tx failed OR not Acked

**ESP_ERR_DPP_INVALID_ATTR**
Encountered invalid DPP Attribute

Type Definitions

typedef enum _dpp_bootstrap_type esp_supp_dpp_bootstrap_t
Types of Bootstrap Methods for DPP.

typedef void (*esp_supp_dpp_event_cb_t)(esp_supp_dpp_event_t evt, void *data)
Callback function for receiving DPP Events from Supplicant.

Callback function will be called with DPP related information.

- **Param evt** DPP event ID
- **Param data** Event data payload

Enumerations

enum _dpp_bootstrap_type
Types of Bootstrap Methods for DPP.

**Values:**

- enumerator **DPP_BOOTSTRAP_QR_CODE**
  QR Code Method

- enumerator **DPP_BOOTSTRAP_PKEX**
  Proof of Knowledge Method

- enumerator **DPP_BOOTSTRAP_NFC_URI**
  NFC URI record Method

enum _esp_supp_dpp_event_t
Types of Callback Events received from DPP Supplicant.

**Values:**

- enumerator **ESP_SUPP_DPP_URI_READY**
  URI is ready through Bootstrapping
Wi-Fi Aware™ (NAN)

Wi-Fi Aware™ or NAN (Neighbor Awareness Networking) is a protocol that allows Wi-Fi devices to discover services in their proximity. Typically, location-based services are based on querying servers for information about the environment and the location knowledge is based on GPS or other location reckoning techniques. However, NAN does not require real-time connection to servers, GPS or other geo-location, but instead uses direct device-to-device Wi-Fi to discover and exchange information. NAN scales effectively in dense Wi-Fi environments and complements the connectivity of Wi-Fi by providing information about people and services in the proximity.

Multiple NAN devices which are in the vicinity form a NAN cluster which allows them to communicate with each other. Devices within a NAN cluster can advertise (Publish method) or look for (Subscribe method) services using NAN Service Discovery protocols. Matching of services is done by service name, once a match is found, a device can either send a message or establish an IPv6 Datapath with the peer.

ESP32-S2 supports Wi-Fi Aware in standalone mode with support for both Service Discovery and Datapath. Wi-Fi Aware is still an evolving protocol. Please refer to Wi-Fi Alliance’s official page on Wi-Fi Aware for more information. Many Android smartphones with Android 8 or higher support Wi-Fi Aware. Refer to Android’s developer guide on Wi-Fi Aware for more information.

Application Example A pair of examples for a Publisher-Subscriber use case: wifi/wifi_aware/nan_publisher and wifi/wifi_aware/nan_subscriber. A user interactive console example to explore full functionality of Wi-Fi Aware: wifi/wifi_aware/nan_console. Please check the README for more details in respective example directories.

API Reference

Header File

- `components/esp_wifi/wifi_apps/include/esp_nan.h`
- This header file can be included with:

```
#include "esp_nan.h"
```
- This header file is a part of the API provided by the `esp_wifi` component. To declare that your component depends on `esp_wifi`, add the following to your `CMakeLists.txt`:

```
REQUIRES esp_wifi
```

or

```
PRIV_REQUIRES esp_wifi
```

Functions

`esp_err_t esp_wifi_nan_start (const wifi_nan_config_t *nan_cfg)`

Start NAN Discovery with provided configuration.

Attention This API should be called after `esp_wifi_init()`.
**Parameters** nan_cfg – NAN related parameters to be configured.

**Returns**
- ESP_OK: succeed
- others: failed

```c
esp_err_t esp_wifi_nan_stop (void)
```

Stop NAN Discovery, end NAN Services and Datapaths.

**Returns**
- ESP_OK: succeed
- others: failed

```c
uint8_t esp_wifi_nan_publish_service (const wifi_nan_publish_cfg_t *publish_cfg, bool ndp_resp_needed)
```

Start Publishing a service to the NAN Peers in vicinity.

**Attention** This API should be called after esp_wifi_nan_start().

**Parameters**
- publish_cfg – Configuration parameters for publishing a service.
- ndp_resp_needed – Setting this true will require user response for every NDP Req using esp_wifi_nan_datapath_resp API.

**Returns**
- non-zero: Publish service identifier
- zero: failed

```c
uint8_t esp_wifi_nan_subscribe_service (const wifi_nan_subscribe_cfg_t *subscribe_cfg)
```

Subscribe for a service within the NAN cluster.

**Attention** This API should be called after esp_wifi_nan_start().

**Parameters** subscribe_cfg – Configuration parameters for subscribing for a service.

**Returns**
- non-zero: Subscribe service identifier
- zero: failed

```c
esp_err_t esp_wifi_nan_send_message (wifi_nan_followup_params_t *fup_params)
```

Send a follow-up message to the NAN Peer with matched service.

**Attention** This API should be called after a NAN service is discovered due to a match.

**Parameters** fup_params – Configuration parameters for sending a Follow-up message.

**Returns**
- ESP_OK: succeed
- others: failed

```c
esp_err_t esp_wifi_nan_cancel_service (uint8_t service_id)
```

Cancel a NAN service.

**Parameters** service_id – Publish/Subscribe service id to be cancelled.

**Returns**
- ESP_OK: succeed
- others: failed
uint8_t esp_wifi_nan_datapath_req (wifi_nan_datapath_req_t *req)

Send NAN Datapath Request to a NAN Publisher with matched service.

**Attention** This API should be called by the Subscriber after a match occurs with a Publisher.

**Parameters**
- req – NAN Datapath Request parameters.

**Returns**
- non-zero NAN Datapath identifier: If NAN datapath req was accepted by publisher
- zero: If NAN datapath req was rejected by publisher or a timeout occurs

esp_err_t esp_wifi_nan_datapath_resp (wifi_nan_datapath_resp_t *resp)

Respond to a NAN Datapath request with Accept or Reject.

**Attention** This API should be called if ndp_auto_accept is not set True by the Publisher and a WIFI_EVENT_NDP_INDICATION event is received due to an incoming NDP request.

**Parameters**
- resp – NAN Datapath Response parameters.

**Returns**
- ESP_OK: succeed
- others: failed

esp_err_t esp_wifi_nan_datapath_end (wifi_nan_datapath_end_req_t *req)

Terminate a NAN Datapath.

**Parameters**
- req – NAN Datapath end request parameters.

**Returns**
- ESP_OK: succeed
- others: failed

void esp_wifi_nan_get_ipv6_linklocal_from_mac (ip6_addr_t*ip6, uint8_t*mac_addr)

Get IPv6 Link Local address using MAC address.

**Parameters**
- ip6 – [out] Derived IPv6 Link Local address.

esp_err_t esp_wifi_nan_get_own_svc_info (uint8_t*own_svc_id, char*svc_name, int*num_peer_records)

brief Get own Service information from Service ID OR Name.

**Attention** If service information is to be fetched from service name, set own_svc_id as zero.

**Parameters**
- own_svc_id – [inout] As input, it indicates Service ID to search for. As output, it indicates Service ID of the service found using Service Name.
- svc_name – [inout] As input, it indicates Service Name to search for. As output, it indicates Service Name of the service found using Service ID.
- num_peer_records – [out] Number of peers discovered by corresponding service.

**Returns**
- ESP_OK: succeed
- ESP_FAIL: failed

esp_err_t esp_wifi_nan_get_peer_records (int*num_peer_records, uint8_t own_svc_id, struct nanpeer_record*peer_record)

brief Get a list of Peers discovered by the given Service.
Parameters

• **num_peer_records** – [inout] As input param, it stores max peers peer_record can hold. As output param, it specifies the actual number of peers this API returns.
• **own_svc_id** – Service ID of own service.
• **peer_record** – [out] Pointer to first peer record.

Returns

• ESP_OK: succeed
• ESP_FAIL: failed

\[
\text{esp_err_t esp_wifi_nan_get_peer_info}(\text{char}* \text{svc_name}, \text{uint8_t}*, \text{peer_mac}, \text{struct nan_peer_record} * \text{peer_info})
\]

brief Find Peer’s Service information using Peer MAC and optionally Service Name.

Parameters

• **svc_name** – Service Name of the published/subscribed service.
• **peer_mac** – Peer’s NAN Management Interface MAC address.
• **peer_info** – [out] Peer’s service information structure.

Returns

• ESP_OK: succeed
• ESP_FAIL: failed

Structures

struct `nan_peer_record`
Parameters of a peer service record

Public Members

\[
\text{uint8_t peer_svc_id}
\]
Identifier of Peer’s service

\[
\text{uint8_t own_svc_id}
\]
Identifier of own service associated with Peer

\[
\text{uint8_t peer_nmi[6]}
\]
Peer’s NAN Management Interface address

\[
\text{uint8_t peer_svc_type}
\]
Peer’s service type (Publish/Subscribe)

\[
\text{uint8_t ndp_id}
\]
Specifies if the peer has any active datapath

\[
\text{uint8_t peer_ndi[6]}
\]
Peer’s NAN Data Interface address, only valid when ndp_id is non-zero

Macros

\[
\text{WIFI\_NAN\_CONFIG\_DEFAULT}()
\]

\[
\text{NDP\_STATUS\_ACCEPTED}
\]

\[
\text{NDP\_STATUS\_REJECTED}
\]
NAN_MAX_PEERS_RECORD

ESP_NAN_PUBLISH

ESP_NAN_SUBSCRIBE

Code examples for the Wi-Fi API are provided in the wifi directory of ESP-IDF examples.
Code examples for ESP-WIFI-MESH are provided in the mesh directory of ESP-IDF examples.

2.4.2 Ethernet

Ethernet

Overview  ESP-IDF provides a set of consistent and flexible APIs to support both internal Ethernet MAC (EMAC) controller and external SPI-Ethernet modules.

This programming guide is split into the following sections:

1. Basic Ethernet Concepts
2. Configure MAC and PHY
3. Connect Driver to TCP/IP Stack
4. Misc Control of Ethernet Driver

Basic Ethernet Concepts  Ethernet is an asynchronous Carrier Sense Multiple Access with Collision Detect (CSMA/CD) protocol/interface. It is generally not well suited for low-power applications. However, with ubiquitous deployment, internet connectivity, high data rates, and limitless-range expandability, Ethernet can accommodate nearly all wired communications.

Normal IEEE 802.3 compliant Ethernet frames are between 64 and 1518 bytes in length. They are made up of five or six different fields: a destination MAC address (DA), a source MAC address (SA), a type/length field, a data payload, an optional padding field and a Cyclic Redundancy Check (CRC). Additionally, when transmitted on the Ethernet medium, a 7-byte preamble field and Start-of-Frame (SOF) delimiter byte are appended to the beginning of the Ethernet packet.

Thus the traffic on the twist-pair cabling appears as shown below:

Preamble and Start-of-Frame Delimiter  The preamble contains seven bytes of $55H$. It allows the receiver to lock onto the stream of data before the actual frame arrives.

The Start-of-Frame Delimiter (SFD) is a binary sequence $10101011$ (as seen on the physical medium). It is sometimes considered to be part of the preamble.

When transmitting and receiving data, the preamble and SFD bytes will be automatically generated or stripped from the packets.

Destination Address  The destination address field contains a 6-byte length MAC address of the device that the packet is directed to. If the Least Significant bit in the first byte of the MAC address is set, the address is a multicast destination. For example, 01-00-00-00-F0-00 and 33-45-67-89-AB-CD are multi-cast addresses, while 00-00-00-00-F0-00 and 32-45-67-89-AB-CD are not.

Packets with multi-cast destination addresses are designed to arrive and be important to a selected group of Ethernet nodes. If the destination address field is the reserved multicast address, i.e., FF-FF-FF-FF-FF-FF, the packet is a broadcast packet and it will be directed to everyone sharing the network. If the Least Significant bit in the first byte
Chapter 2. API Reference

Fig. 3: Ethernet Data Frame Format

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble (7 Bytes)</td>
<td>Start-of-Frame Delimiter (1 Byte)</td>
<td>Destination Address (6 Bytes)</td>
<td>Source Address (6 Bytes)</td>
<td>Type / Length (2 Bytes)</td>
<td>Payload (0 ~ 1500 Bytes)</td>
<td>Pad (if necessary)</td>
</tr>
<tr>
<td>Frame Check Sequence (4 Bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

of the MAC address is clear, the address is a unicast address and will be designed for usage by only the addressed node.

Normally the EMAC controller incorporates receive filters which can be used to discard or accept packets with multicast, broadcast and/or unicast destination addresses. When transmitting packets, the host controller is responsible for writing the desired destination address into the transmit buffer.

**Source Address**  The source address field contains a 6-byte length MAC address of the node which created the Ethernet packet. Users of Ethernet must generate a unique MAC address for each controller used. MAC addresses consist of two portions. The first three bytes are known as the Organizationally Unique Identifier (OUI). OUIs are distributed by the IEEE. The last three bytes are address bytes at the discretion of the company that purchased the OUI. For more information about MAC Address used in ESP-IDF, please see MAC Address Allocation.

When transmitting packets, the assigned source MAC address must be written into the transmit buffer by the host controller.

**Type/Length**  The type/length field is a 2-byte field. If the value in this field is <= 1500 (decimal), it is considered a length field and it specifies the amount of non-padding data which follows in the data field. If the value is >= 1536, it represents the protocol the following packet data belongs to. The followings are the most common type values:

- IPv4 = 0800H
- IPv6 = 86DDH
- ARP = 0806H

Users implementing proprietary networks may choose to treat this field as a length field, while applications implementing protocols such as the Internet Protocol (IP) or Address Resolution Protocol (ARP), should program this field with the appropriate type defined by the protocol’s specification when transmitting packets.
Payload  The payload field is a variable length field, anywhere from 0 to 1500 bytes. Larger data packets violates Ethernet standards and will be dropped by most Ethernet nodes.

This field contains the client data, such as an IP datagram.

Padding and FCS  The padding field is a variable length field added to meet the IEEE 802.3 specification requirements when small data payloads are used.

The DA, SA, type, payload, and padding of an Ethernet packet must be no smaller than 60 bytes in total. If the required 4-byte FCS field is added, packets must be no smaller than 64 bytes. If the payload field is less than 46-byte long, a padding field is required.

The FCS field is a 4-byte field that contains an industry-standard 32-bit CRC calculated with the data from the DA, SA, type, payload, and padding fields. Given the complexity of calculating a CRC, the hardware normally automatically generates a valid CRC and transmit it. Otherwise, the host controller must generate the CRC and place it in the transmit buffer.

Normally, the host controller does not need to concern itself with padding and the CRC which the hardware EMAC will also be able to automatically generate when transmitting and verify when receiving. However, the padding and CRC fields will be written into the receive buffer when packets arrive, so they may be evaluated by the host controller if needed.

Note: Besides the basic data frame described above, there are two other common frame types in 10/100 Mbps Ethernet: control frames and VLAN-tagged frames. They are not supported in ESP-IDF.

Configure MAC and PHY  The Ethernet driver is composed of two parts: MAC and PHY.

You need to set up the necessary parameters for MAC and PHY respectively based on your Ethernet board design, and then combine the two together to complete the driver installation.

Configuration for MAC is described in `eth_mac_config_t`, including:

- `eth_mac_config_t::sw_reset_timeout_ms`: software reset timeout value, in milliseconds. Typically, MAC reset should be finished within 100 ms.
- `eth_mac_config_t::rx_task_stack_size` and `eth_mac_config_t::rx_task_prio`: the MAC driver creates a dedicated task to process incoming packets. These two parameters are used to set the stack size and priority of the task.
- `eth_mac_config_t::flags`: specifying extra features that the MAC driver should have, it could be useful in some special situations. The value of this field can be OR’ed with macros prefixed with `ETH_MAC_FLAG_`. For example, if the MAC driver should work when the cache is disabled, then you should configure this field with `ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE`.

Configuration for PHY is described in `eth_phy_config_t`, including:

- `eth_phy_config_t::phy_addr`: multiple PHY devices can share the same SMI bus, so each PHY needs a unique address. Usually, this address is configured during hardware design by pulling up/down some PHY strapping pins. You can set the value from 0 to 15 based on your Ethernet board. Especially, if the SMI bus is shared by only one PHY device, setting this value to -1 can enable the driver to detect the PHY address automatically.
- `eth_phy_config_t::reset_timeout_ms`: reset timeout value, in milliseconds. Typically, PHY reset should be finished within 100 ms.
- `eth_phy_config_t::autonego_timeout_ms`: auto-negotiation timeout value, in milliseconds. The Ethernet driver starts negotiation with the peer Ethernet node automatically, to determine to duplex and speed mode. This value usually depends on the ability of the PHY device on your board.
- `eth_phy_config_t::reset_gpio_num`: if your board also connects the PHY reset pin to one of the GPIO, then set it here. Otherwise, set this field to -1.
ESP-IDF provides a default configuration for MAC and PHY in macro `ETH_MAC_DEFAULT_CONFIG` and `ETH_PHY_DEFAULT_CONFIG`.

Create MAC and PHY Instance  The Ethernet driver is implemented in an Object-Oriented style. Any operation on MAC and PHY should be based on the instance of the two.

SPI-Ethernet Module

```c
eth_mac_config_t mac_config = ETH_MAC_DEFAULT_CONFIG();  // apply default MAC configuration
eth_phy_config_t phy_config = ETH_PHY_DEFAULT_CONFIG();  // apply default PHY configuration
phy_config.phy_addr = CONFIG_EXAMPLE_ETH_PHY_ADDR;  // alter the PHY address according to your board design
phy_config.reset_gpio_num = CONFIG_EXAMPLE_ETH_PHY_RST_GPIO;  // alter the GPIO used for PHY reset
spi_device_handle_t spi_handle = NULL;
spi_bus_config_t buscfg = {
    .miso_io_num = CONFIG_EXAMPLE_ETH_SPI_MISO_GPIO,
    .mosi_io_num = CONFIG_EXAMPLE_ETH_SPI_MOSI_GPIO,
    .sclk_io_num = CONFIG_EXAMPLE_ETH_SPI_SCLK_GPIO,
    .quadwp_io_num = -1,
    .quadhd_io_num = -1,
};
ESP_ERROR_CHECK(spi_bus_initialize(CONFIG_EXAMPLE_ETH_SPI_HOST, &buscfg, 1));
eth_dm9051_config_t dm9051_config = ETH_DM9051_DEFAULT_CONFIG(CONFIG_EXAMPLE_ETH_SPI_HOST, &spi_devcfg);
dm9051_config.int_gpio_num = CONFIG_EXAMPLE_ETH_SPI_INT_GPIO;
esp_eth_mac_t *mac = esp_eth_mac_new_dm9051(&dm9051_config, &mac_config);
esp_eth_phy_t *phy = esp_eth_phy_new_dm9051(&phy_config);
```

Note:

- When creating MAC and PHY instances for SPI-Ethernet modules (e.g., DM9051), the constructor function must have the same suffix (e.g., `esp_eth_mac_new_dm9051` and `esp_eth_phy_new_dm9051`). This is because we don’t have other choices but the integrated PHY.
- The SPI device configuration (i.e., `spi_device_interface_config_t`) may slightly differ for other Ethernet modules or to meet SPI timing on specific PCB. Please check out your module’s specs and the examples in ESP-IDF.

Install Driver  To install the Ethernet driver, we need to combine the instance of MAC and PHY and set some additional high-level configurations (i.e., not specific to either MAC or PHY) in `esp_eth_config_t`:

- `esp_eth_config_t::mac`: instance that created from MAC generator (e.g., `esp_eth_mac_new_dm9051()`).
- `esp_eth_config_t::phy`: instance that created from PHY generator (e.g., `esp_eth_phy_new_ip101()`).
- `esp_eth_config_t::check_link_period_ms`: Ethernet driver starts an OS timer to check the link status periodically, this field is used to set the interval, in milliseconds.
Chapter 2. API Reference

- `esp_eth_config_t::stack_input`: In most Ethernet IoT applications, any Ethernet frame received by a driver should be passed to the upper layer (e.g., TCP/IP stack). This field is set to a function that is responsible to deal with the incoming frames. You can even update this field at runtime via function `esp_eth_update_input_path()` after driver installation.
- `esp_eth_config_t::on_lowlevel_init_done` and `esp_eth_config_t::on_lowlevel_deinit_done`: These two fields are used to specify the hooks which get invoked when low-level hardware has been initialized or de-initialized.

ESP-IDF provides a default configuration for driver installation in macro `ETH_DEFAULT_CONFIG`.

```c
esp_eth_config_t config = ETH_DEFAULT_CONFIG(mac, phy); // apply default driver configuration
esp_eth_handle_t eth_handle = NULL; // after the driver is installed, we will get the handle of the driver
esp_eth_driver_install(&config, &eth_handle); // install driver
```

The Ethernet driver also includes an event-driven model, which sends useful and important events to user space. We need to initialize the event loop before installing the Ethernet driver. For more information about event-driven programming, please refer to ESP Event.

```c
/** Event handler for Ethernet events */
static void eth_event_handler(void *arg, esp_event_base_t event_base, int32_t event_id, void *event_data)
{
    uint8_t mac_addr[6] = {0};
    /* we can get the ethernet driver handle from event data */
    esp_eth_handle_t eth_handle = *(esp_eth_handle_t *)event_data;

    switch (event_id) {
    case ETHERNET_EVENT_CONNECTED:
        esp_eth_ioctl(eth_handle, ETH_CMD_G_MAC_ADDR, mac_addr);
        ESP_LOGI(TAG, "Ethernet Link Up");
        ESP_LOGI(TAG, "Ethernet HW Addr %02x:%02x:%02x:%02x:%02x:%02x",
                 mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4], mac_addr[5]);
        break;
    case ETHERNET_EVENT_DISCONNECTED:
        ESP_LOGI(TAG, "Ethernet Link Down");
        break;
    case ETHERNET_EVENT_START:
        ESP_LOGI(TAG, "Ethernet Started");
        break;
    case ETHERNET_EVENT_STOP:
        ESP_LOGI(TAG, "Ethernet Stopped");
        break;
    default:
        break;
    }
}
```

```c
esp_event_loop_create_default(); // create a default event loop that runs in the background
esp_event_handler_register(ETH_EVENT, ETH_EVENT_ANY_ID, &eth_event_handler, NULL); // register Ethernet event handler (to deal with user-specific stuff when events like link up/down happened)
```

**Start Ethernet Driver**  After driver installation, we can start Ethernet immediately.

```c
esp_eth_start(eth_handle); // start Ethernet driver state machine
```
Chapter 2. API Reference

**Connect Driver to TCP/IP Stack**  Up until now, we have installed the Ethernet driver. From the view of OSI (Open System Interconnection), we are still on level 2 (i.e., Data Link Layer). While we can detect link up and down events and gain MAC address in user space, it is infeasible to obtain the IP address, let alone send an HTTP request. The TCP/IP stack used in ESP-IDF is called LwIP. For more information about it, please refer to *LwIP*.

To connect the Ethernet driver to TCP/IP stack, follow these three steps:

1. Create a network interface for the Ethernet driver
2. Attach the network interface to the Ethernet driver
3. Register IP event handlers

For more information about the network interface, please refer to *Network Interface*.

```c
/** Event handler for IP_EVENT_ETH_GOT_IP */
static void got_ip_event_handler(void *arg, esp_event_base_t event_base, int32_t event_id, void *event_data)
{
    ip_event_got_ip_t *event = (ip_event_got_ip_t *) event_data;
    const esp_netif_ip_info_t *ip_info = &event->ip_info;

    ESP_LOGI(TAG, "Ethernet Got IP Address");
    ESP_LOGI(TAG, "~~~~~~~~~~~");
    ESP_LOGI(TAG, "ETHIP: IPSTR, IP2STR(ip_info->ip));
    ESP_LOGI(TAG, "ETHMASK: IPSTR, IP2STR(ip_info->netmask));
    ESP_LOGI(TAG, "ETHGW: IPSTR, IP2STR(ip_info->gw));
    ESP_LOGI(TAG, "~~~~~~~~~~~");
}
```
Flow Control  Ethernet on MCU usually has a limitation in the number of frames it can handle during network congestion, because of the limitation in RAM size. A sending station might be transmitting data faster than the peer end can accept it. The ethernet flow control mechanism allows the receiving node to signal the sender requesting the suspension of transmissions until the receiver catches up. The magic behind that is the pause frame, which was defined in IEEE 802.3x.

Pause frame is a special Ethernet frame used to carry the pause command, whose EtherType field is 0x8808, with the Control opcode set to 0x0001. Only stations configured for full-duplex operation may send pause frames. When a station wishes to pause the other end of a link, it sends a pause frame to the 48-bit reserved multicast address of 01-80-c2-00-00-01. The pause frame also includes the period of pause time being requested, in the form of a two-byte integer, ranging from 0 to 65535.

After the Ethernet driver installation, the flow control feature is disabled by default. You can enable it by:

```c
bool flow_ctrl_enable = true;
esp_eth_ioctl(eth_handle, ETH_CMD_S_FLOW_CTRL, &flow_ctrl_enable);
```

One thing that should be kept in mind is that the pause frame ability is advertised to the peer end by PHY during auto-negotiation. The Ethernet driver sends a pause frame only when both sides of the link support it.

Application Examples

- Ethernet basic example: ethernet/basic
- Ethernet iperf example: ethernet/iperf
- Ethernet to Wi-Fi AP "router": network/eth2ap
- Wi-Fi station to Ethernet "bridge": network/sta2eth
- Most protocol examples should also work for Ethernet: protocols

Advanced Topics

Custom PHY Driver  There are multiple PHY manufacturers with wide portfolios of chips available. The ESP-IDF already supports several PHY chips however one can easily get to a point where none of them satisfies the user’s actual needs due to price, features, stock availability, etc.

Luckily, a management interface between EMAC and PHY is standardized by IEEE 802.3 in Section 22.2.4 Management Functions. It defines provisions of the so-called “MII Management Interface” to control the PHY and gather status from the PHY. A set of management registers is defined to control chip behavior, link properties, auto-negotiation configuration, etc. This basic management functionality is addressed by esp_eth/src/esp_eth_phy_802_3.c in ESP-IDF and so it makes the creation of a new custom PHY chip driver quite a simple task.

Note:  Always consult with PHY datasheet since some PHY chips may not comply with IEEE 802.3, Section 22.2.4. It does not mean you are not able to create a custom PHY driver, but it just requires more effort. You will have to define all PHY management functions.

The majority of PHY management functionality required by the ESP-IDF Ethernet driver is covered by the esp_eth/src/esp_eth_phy_802_3.c. However, the following may require developing chip-specific management functions:

- Link status which is almost always chip-specific
• Chip initialization, even though not strictly required, should be customized to at least ensure that the expected chip is used
• Chip-specific features configuration

Steps to create a custom PHY driver:
1. Define vendor-specific registry layout based on the PHY datasheet. See esp_eth/src/esp_eth_phy_ip101.c as an example.
2. Prepare derived PHY management object info structure which:
   • must contain at least parent IEEE 802.3 phy_802_3_t object
   • optionally contain additional variables needed to support non-IEEE 802.3 or customized functionality. See esp_eth/src/esp_eth_phy ksz80xx.c as an example.
3. Define chip-specific management call-back functions.
4. Initialize parent IEEE 802.3 object and re-assign chip-specific management call-back functions.

Once you finish the new custom PHY driver implementation, consider sharing it among other users via IDF Component Registry.

API Reference

Header File
• components/esp_eth/include/esp_eth.h
  • This header file can be included with:

```
#include "esp_eth.h"
```

• This header file is a part of the API provided by the esp_eth component. To declare that your component depends on esp_eth, add the following to your CMakeLists.txt:

```
REQUIRES esp_eth
```

or

```
PRIV_REQUIRES esp_eth
```

Header File
• components/esp_eth/include/esp_eth_driver.h
  • This header file can be included with:

```
#include "esp_eth_driver.h"
```

• This header file is a part of the API provided by the esp_eth component. To declare that your component depends on esp_eth, add the following to your CMakeLists.txt:

```
REQUIRES esp_eth
```

or

```
PRIV_REQUIRES esp_eth
```

Functions

`esp_err_t esp_eth_driver_install(const esp_eth_config_t *config, esp_eth_handle_t *out_hdl)`

Install Ethernet driver.

Parameters
• `config` [in] configuration of the Ethernet driver
• `out_hdl` [out] handle of Ethernet driver

Returns
• ESP_OK: install esp_eth driver successfully
Chapter 2. API Reference

- 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

**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.

**Parameters**

- **hdl** - [in] handle of Ethernet driver

**Returns**

- 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

**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).

**Parameters**

- **hdl** - [in] handle of Ethernet driver

**Returns**

- 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

**esp_err_t esp_eth_stop (esp_eth_handle_t hdl)**

Stop Ethernet driver.

**Note:** This function does the opposite operation of esp_eth_start.

**Parameters**

- **hdl** - [in] handle of Ethernet driver

**Returns**

- 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

**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.

Parameters
- 
  **hdl** – [in] handle of Ethernet driver
- 
  **stack_input** – [in] function pointer, which does the actual process on incoming packets
- 
  **priv** – [in] private resource, which gets passed to stack_input callback without any modification

Returns
- 
  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

```c
esp_err_t esp_eth_transmit(esp_eth_handle_t hdl, void *buf, size_t length)
```
General Transmit.

Parameters
- 
  **hdl** – [in] handle of Ethernet driver
- 
  **buf** – [in] buffer of the packet to transfer
- 
  **length** – [in] length of the buffer to transfer

Returns
- 
  ESP_OK: transmit frame buffer successfully
- 
  ESP_ERR_INVALID_ARG: transmit frame buffer failed because of some invalid argument
- 
  ESP_ERR_INVALID_STATE: invalid driver state (e.i. driver is not started)
- 
  ESP_ERR_TIMEOUT: transmit frame buffer failed because HW was not get available in predefined period
- 
  ESP_FAIL: transmit frame buffer failed because some other error occurred

```c
esp_err_t esp_eth_transmit_vargs(esp_eth_handle_t hdl, uint32_t argc, ...)
```
Special Transmit with variable number of arguments.

Parameters
- 
  **hdl** – [in] handle of Ethernet driver
- 
  **argc** – [in] number variable arguments
- 
  **...** – variable arguments

Returns
- 
  ESP_OK: transmit successfull
- 
  ESP_ERR_INVALID_STATE: invalid driver state (e.i. driver is not started)
- 
  ESP_ERR_TIMEOUT: transmit frame buffer failed because HW was not get available in predefined period
- 
  ESP_FAIL: transmit frame buffer failed because some other error occurred

```c
esp_err_t esp_eth_ioctl(esp_eth_handle_t hdl, esp_eth_io_cmd_t cmd, void *data)
```
Misc IO function of Ethirteen driver.

The following common IO control commands are supported:

- 
  **ETH_CMD_S_MAC_ADDR** sets Ethernet interface MAC address. data argument is pointer to MAC address buffer with expected size of 6 bytes.
- 
  **ETH_CMD_G_MAC_ADDR** gets Ethernet interface MAC address. data argument is pointer to a buffer to which MAC address is to be copied. The buffer size must be at least 6 bytes.
- 
  **ETH_CMD_S_PHY_ADDR** sets PHY address in range of <0-31>. data argument is pointer to memory of uint32_t datatype from where the configuration option is read.
- 
  **ETH_CMD_G_PHY_ADDR** gets PHY address. data argument is pointer to memory of uint32_t datatype to which the PHY address is to be stored.
• **ETH_CMD_S_AUTONEGO** enables or disables Ethernet link speed and duplex mode autonegotiation. The `data` argument is pointer to memory of bool datatype from which the configuration option is read. Preconditions: Ethernet driver needs to be stopped.

• **ETH_CMD_G_AUTONEGO** gets current configuration of the Ethernet link speed and duplex mode autonegotiation. The `data` argument is pointer to memory of bool datatype to which the current configuration is to be stored.

• **ETH_CMD_S_SPEED** sets the Ethernet link speed. The `data` argument is pointer to memory of `eth_speed_t` datatype from which the configuration option is read. Preconditions: Ethernet driver needs to be stopped and auto-negotiation disabled.

• **ETH_CMD_G_SPEED** gets current Ethernet link speed. The `data` argument is pointer to memory of `eth_speed_t` datatype to which the current configuration is stored.

• **ETH_CMD_S_PROMISCUOUS** sets/resets Ethernet interface promiscuous mode. The `data` argument is pointer to memory of bool datatype from which the configuration option is read.

• **ETH_CMD_S_FLOW_CTRL** sets/resets Ethernet interface flow control. The `data` argument is pointer to memory of bool datatype from which the configuration option is read.

• **ETH_CMD_S_DUPLEX_MODE** sets the Ethernet duplex mode. The `data` argument is pointer to memory of `eth_duplex_t` datatype from which the configuration option is read. Preconditions: Ethernet driver needs to be stopped and auto-negotiation disabled.

• **ETH_CMD_G_DUPLEX_MODE** gets current Ethernet link duplex mode. The `data` argument is pointer to memory of `eth_duplex_t` datatype to which the duplex mode is to be stored.

• **ETH_CMD_S_PHY_LOOPBACK** sets/resets PHY to/from loopback mode. The `data` argument is pointer to memory of bool datatype from which the configuration option is read.

• Note that additional control commands may be available for specific MAC or PHY chips. Please consult specific MAC or PHY documentation or driver code.

### Parameters
- **hdl** [in] handle of Ethernet driver
- **cmd** [in] IO control command
- **data** [inout] address of data for set command or address where to store the data when used with get command

### Returns
- 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
- ESP_ERR_NOT_SUPPORTED: requested feature is not supported

**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.

**Parameters**
- **hdl** [in] handle of Ethernet driver

**Returns**
- ESP_OK: increase reference successfully
- ESP_ERR_INVALID_ARG: increase reference failed because of some invalid argument

**esp_err_t esp_eth_decrease_reference (esp_eth_handle_t hdl)**

Decrease Ethernet driver reference.

**Parameters**
- **hdl** [in] handle of Ethernet driver

**Returns**
- ESP_OK: increase reference successfully
Chapter 2. API Reference

• ESP_ERR_INVALID_ARG: increase reference failed because of some invalid argument

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.

Param eth_handle [in] handle of Ethernet driver
Param buffer [in] frame buffer that will get input to upper stack
Param length [in] length of the frame buffer
Return
• ESP_OK: input frame buffer to upper stack successfully
• ESP_FAIL: error occurred when inputting buffer to upper stack

esp_err_t (*on_lowlevel_init_done)(esp_eth_handle_t eth_handle)
Callback function invoked when lowlevel initialization is finished.

Param eth_handle [in] handle of Ethernet driver
Return
• ESP_OK: process extra lowlevel initialization successfully
• ESP_FAIL: error occurred when processing extra lowlevel initialization

esp_err_t (*on_lowlevel_deinit_done)(esp_eth_handle_t eth_handle)
Callback function invoked when lowlevel deinitialization is finished.

Param eth_handle [in] handle of Ethernet driver
Return
• ESP_OK: process extra lowlevel deinitialization successfully
• ESP_FAIL: error occurred when processing extra lowlevel deinitialization

esp_err_t (*read_phy_reg)(esp_eth_handle_t eth_handle, uint32_t phy_addr, uint32_t phy_reg, uint32_t *reg_value)
Read PHY register.

Note: Usually the PHY register read/write function is provided by MAC (SMI interface), but if the PHY device is managed by other interface (e.g. I2C), then user needs to implement the corresponding read/write. Setting this to NULL means your PHY device is managed by MAC’s SMI interface.
Chapter 2. API Reference

Param phy_addr [in] PHY chip address (0~31)
Param phy_reg [in] PHY register index code
Param reg_value [out] PHY register value

Return
  • ESP_OK: read PHY register successfully
  • ESP_ERR_INVALID_ARG: read PHY register failed because of invalid argument
  • ESP_ERR_TIMEOUT: read PHY register failed because of timeout
  • ESP_FAIL: read PHY register failed because some other error occurred

esp_err_t (*write_phy_reg)(esp_eth_handle_t eth_handle, uint32_t phy_addr, uint32_t phy_reg, uint32_t reg_value)

Write PHY register.

Note: Usually the PHY register read/write function is provided by MAC (SMI interface), but if the PHY device is managed by other interface (e.g. I2C), then user needs to implement the corresponding read/write. Setting this to NULL means your PHY device is managed by MAC’s SMI interface.

Param eth_handle [in] handle of Ethernet driver
Param phy_addr [in] PHY chip address (0~31)
Param phy_reg [in] PHY register index code
Param reg_value [in] PHY register value

Return
  • ESP_OK: write PHY register successfully
  • ESP_ERR_INVALID_ARG: read PHY register failed because of invalid argument
  • ESP_ERR_TIMEOUT: write PHY register failed because of timeout
  • ESP_FAIL: write PHY register failed because some other error occurred

struct esp_eth_phy_reg_rw_data_t
Data structure to Read/Write PHY register via ioctl API.

Public Members

uint32_t reg_addr
  PHY register address

uint32_t *reg_value_p
  Pointer to a memory where the register value is read/written

Macros

ETH_DEFAULT_CONFIG (emac, ephy)
Default configuration for Ethernet driver.

Type Definitions

typedef void *esp_eth_handle_t
Handle of Ethernet driver.

Enumerations
enum esp_eth_io_cmd_t
    Command list for ioctl API.
    
    Values:
    
    enumerator ETH_CMD_G_MAC_ADDR
        Get MAC address
    
    enumerator ETH_CMD_S_MAC_ADDR
        Set MAC address
    
    enumerator ETH_CMD_G_PHY_ADDR
        Get PHY address
    
    enumerator ETH_CMD_S_PHY_ADDR
        Set PHY address
    
    enumerator ETH_CMD_G_AUTONEGO
        Get PHY Auto Negotiation
    
    enumerator ETH_CMD_S_AUTONEGO
        Set PHY Auto Negotiation
    
    enumerator ETH_CMD_G_SPEED
        Get Speed
    
    enumerator ETH_CMD_S_SPEED
        Set Speed
    
    enumerator ETH_CMD_S_PROMISCUOUS
        Set promiscuous mode
    
    enumerator ETH_CMD_S_FLOW_CTRL
        Set flow control
    
    enumerator ETH_CMD_G_DUPLEX_MODE
        Get Duplex mode
    
    enumerator ETH_CMD_S_DUPLEX_MODE
        Set Duplex mode
    
    enumerator ETH_CMD_S_PHY_LOOPBACK
        Set PHY loopback
    
    enumerator ETH_CMD_READ_PHY_REG
        Read PHY register
    
    enumerator ETH_CMD_WRITE_PHY_REG
        Write PHY register
enumerator ETH_CMD_CUSTOM_MAC_CMDS

enumerator ETH_CMD_CUSTOM_PHY_CMDS

Header File
- components/esp_eth/include/esp_eth_com.h
- This header file can be included with:
  
  ```
  #include "esp_eth_com.h"
  ```
- This header file is a part of the API provided by the esp_eth component. To declare that your component depends on esp_eth, add the following to your CMakeLists.txt:
  
  ```
  REQUIRES esp_eth
  ```
or
  ```
  PRIV_REQUIRES esp_eth
  ```

Structures

struct esp_eth_mediator_s
  Ethernet mediator.

Public Members

```c
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.

- **Param**
  - eth [in] mediator of Ethernet driver
  - phy_addr [in] PHY Chip address (0~31)
  - phy_reg [in] PHY register index code
  - reg_value [out] PHY register value

- **Return**
  - ESP_OK: read PHY register successfully
  - ESP_FAIL: read PHY register failed because some error occurred

```c
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.

- **Param**
  - eth [in] mediator of Ethernet driver
  - phy_addr [in] PHY Chip address (0~31)
  - phy_reg [in] PHY register index code
  - reg_value [in] PHY register value

- **Return**
  - ESP_OK: write PHY register successfully
  - ESP_FAIL: write PHY register failed because some error occurred

```c
esp_err_t (*stack_input)(esp_eth_mediator_t *eth, uint8_t *buffer, uint32_t length)
```
Deliver packet to upper stack.

- **Param**
  - eth [in] mediator of Ethernet driver
  - buffer [in] packet buffer
Chapter 2. API Reference

Param length [in] length of the packet

Return
• ESP_OK: deliver packet to upper stack successfully
• ESP_FAIL: deliver packet failed because some error occurred

```c
esp_err_t (*on_state_changed)(esp_eth_mediator_t *eth, esp_eth_state_t state, void *args)
```

Callback on Ethernet state changed.

Param eth [in] mediator of Ethernet driver
Param state [in] new state
Param args [in] optional argument for the new state

Return
• ESP_OK: process the new state successfully
• ESP_FAIL: process the new state failed because some error occurred

Type Definitions
typedef struct esp_eth_mediator_s esp_eth_mediator_t

Ethernet mediator.

Enumerations
enum esp_eth_state_t

Ethernet driver state.

Values:

enumerator ETH_STATE_LLINIT
Lowlevel init done

enumerator ETH_STATE_DEINIT
Deinit done

enumerator ETH_STATE_LINK
Link status changed

enumerator ETH_STATE_SPEED
Speed updated

enumerator ETH_STATE_DUPLEX
Duplex updated

enumerator ETH_STATE_PAUSE
Pause ability updated

enum eth_event_t

Ethernet event declarations.

Values:

enumerator ETHERNET_EVENT_START
Ethernet driver start
enumerator **ETHERNET_EVENT_STOP**
  Ethernet driver stop

enumerator **ETHERNET_EVENT_CONNECTED**
  Ethernet got a valid link

enumerator **ETHERNET_EVENT_DISCONNECTED**
  Ethernet lost a valid link

**Header File**

- components/esp_eth/include/esp_eth_mac.h
- This header file can be included with:
  ```
  #include "esp_eth_mac.h"
  ```
- This header file is a part of the API provided by the esp_eth component. To declare that your component depends on esp_eth, add the following to your CMakeLists.txt:
  ```
  REQUIRE esp_eth
  or
  PRIV_REQUIRES esp_eth
  ```

**Unions**

union **eth_mac_clock_config_t**
  #include <esp_eth_mac.h> Ethernet MAC Clock Configuration.

**Public Members**

struct **eth_mac_clock_config_t**::[anonymous] **mii**
  EMAC MII Clock Configuration

**emac_rmii_clock_mode_t clock_mode**
  RMII Clock Mode Configuration

**emac_rmii_clock_gpio_t clock_gpio**
  RMII Clock GPIO Configuration

struct **eth_mac_clock_config_t**::[anonymous] **rmii**
  EMAC RMII Clock Configuration

**Structures**

struct **esp_eth_mac_s**
  Ethernet MAC.
Public Members

```c
esp_err_t (*set_mediator)(esp_eth_mac_t *mac, esp_eth_mediator_t *eth)
```

Set mediator for Ethernet MAC.

- **Param mac** [in] Ethernet MAC instance
- **Param eth** [in] Ethernet mediator

**Return**
- ESP_OK: set mediator for Ethernet MAC successfully
- ESP_ERR_INVALID_ARG: set mediator for Ethernet MAC failed because of invalid argument

```c
esp_err_t (*init)(esp_eth_mac_t *mac)
```

Initialize Ethernet MAC.

- **Param mac** [in] Ethernet MAC instance

**Return**
- ESP_OK: initialize Ethernet MAC successfully
- ESP_FAIL: initialize Ethernet MAC failed because some other error occurred

```c
esp_err_t (*deinit)(esp_eth_mac_t *mac)
```

Deinitialize Ethernet MAC.

- **Param mac** [in] Ethernet MAC instance

**Return**
- ESP_OK: deinitialize Ethernet MAC successfully
- ESP_FAIL: deinitialize Ethernet MAC failed because some other error occurred

```c
esp_err_t (*start)(esp_eth_mac_t *mac)
```

Start Ethernet MAC.

- **Param mac** [in] Ethernet MAC instance

**Return**
- ESP_OK: start Ethernet MAC successfully
- ESP_FAIL: start Ethernet MAC failed because some other error occurred

```c
esp_err_t (*stop)(esp_eth_mac_t *mac)
```

Stop Ethernet MAC.

- **Param mac** [in] Ethernet MAC instance

**Return**
- ESP_OK: stop Ethernet MAC successfully
- ESP_FAIL: stop Ethernet MAC failed because some other error occurred

```c
esp_err_t (*transmit)(esp_eth_mac_t *mac, uint8_t *buf, uint32_t length)
```

Transmit packet from Ethernet MAC.

**Note:** Returned error codes may differ for each specific MAC chip.

- **Param mac** [in] Ethernet MAC instance
- **Param buf** [in] packet buffer to transmit
- **Param length** [in] length of packet

**Return**
- ESP_OK: transmit packet successfully
Chapter 2. API Reference

- ESP_ERR_INVALID_SIZE: number of actually sent bytes differs to expected
- ESP_FAIL: transmit packet failed because some other error occurred

```c
esp_err_t (*transmit_vargs)(esp_eth_mac_t *mac, uint32_t argc, va_list args)
```
Transmit packet from Ethernet MAC constructed with special parameters at Layer 2.

**Note:** Typical intended use case is to make possible to construct a frame from multiple higher layer buffers without a need of buffer reallocations. However, other use cases are not limited.

**Note:** Returned error codes may differ for each specific MAC chip.

```c
Param mac [in] Ethernet MAC instance
Param argc [in] number variable arguments
Param args [in] variable arguments
```

**Return**
- ESP_OK: transmit packet successfully
- ESP_ERR_INVALID_SIZE: number of actually sent bytes differs to expected
- ESP_FAIL: transmit packet failed because some other error occurred

```c
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.

```c
Param mac [in] Ethernet MAC instance
Param buf [out] packet buffer which will preserve the received frame
Param length [out] length of the received packet
```

**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

```c
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.

```c
Param mac [in] Ethernet MAC instance
Param phy_addr [in] PHY chip address (0~31)
Param phy_reg [in] PHY register index code
Param reg_value [out] PHY register value
```

**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
Chapter 2. API Reference

- ESP_ERR_TIMEOUT: read PHY register failed because of timeout
- ESP_FAIL: read PHY register failed because some other error occurred

```c
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.

- **Param mac** [in] Ethernet MAC instance
- **Param phy_addr** [in] PHY chip address (0~31)
- **Param phy_reg** [in] PHY register index code
- **Param reg_value** [in] PHY register value

**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

```c
esp_err_t (*set_addr)(esp_eth_mac_t *mac, uint8_t *addr)
```

Set MAC address.

- **Param mac** [in] Ethernet MAC instance
- **Param addr** [in] 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

```c
esp_err_t (*get_addr)(esp_eth_mac_t *mac, uint8_t *addr)
```

Get MAC address.

- **Param mac** [in] Ethernet MAC instance
- **Param addr** [out] 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

```c
esp_err_t (*set_speed)(esp_eth_mac_t *mac, eth_speed_t speed)
```

Set speed of MAC.

- **Param mac** [in] Ethernet MAC instance
- **Param speed** [in] MAC speed

**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

```c
esp_err_t (*set_duplex)(esp_eth_mac_t *mac, eth_duplex_t duplex)
```

Set duplex mode of MAC.

- **Param mac** [in] Ethernet MAC instance
- **Param duplex** [in] MAC duplex

**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
**esp_err_t** (*set_link*)(**esp_eth_mac_t** *mac, **eth_link_t** link)
Set link status of MAC.

**Param**
- **mac** [in] Ethernet MAC instance
- **link** [in] Link status

**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

**esp_err_t** (*set_promiscuous*)(**esp_eth_mac_t** *mac, **bool** enable)
Set promiscuous of MAC.

**Param**
- **mac** [in] Ethernet MAC instance
- **enable** [in] set true to enable promiscuous mode; set false to disable promiscuous mode

**Return**
- ESP_OK: set promiscuous mode successfully
- ESP_FAIL: set promiscuous mode failed because some error occurred

**esp_err_t** (*enable_flow_ctrl*)(**esp_eth_mac_t** *mac, **bool** enable)
Enable flow control on MAC layer or not.

**Param**
- **mac** [in] Ethernet MAC instance
- **enable** [in] set true to enable flow control; set false to disable flow control

**Return**
- ESP_OK: set flow control successfully
- ESP_FAIL: set flow control failed because some error occurred

**esp_err_t** (*set_peer_pause_ability*)(**esp_eth_mac_t** *mac, **uint32_t** ability)
Set the PAUSE ability of peer node.

**Param**
- **mac** [in] Ethernet MAC instance
- **ability** [in] zero indicates that pause function is supported by link partner; non-zero indicates that pause function is not supported by link partner

**Return**
- ESP_OK: set peer pause ability successfully
- ESP_FAIL: set peer pause ability failed because some other error occurred

**esp_err_t** (*custom_ioctl*)(**esp_eth_mac_t** *mac, **uint32_t** cmd, **void** *data)
Custom IO function of MAC driver. This function is intended to extend common options of esp_eth_ioctl to cover specifics of MAC chip.

**Note:** This function may not be assigned when the MAC chip supports only most common set of configuration options.

**Param**
- **mac** [in] Ethernet MAC instance
- **cmd** [in] IO control command
- **data** [inout] address of data for set command or address where to store the data when used with get command

**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
- ESP_ERR_NOT_SUPPORTED: requested feature is not supported
esp_err_t (*del)(esp_eth_mac_t *mac)

Free memory of Ethernet MAC.

Param mac [in] Ethernet MAC instance

Return

- ESP_OK: free Ethernet MAC instance successfully
- ESP_FAIL: free Ethernet MAC instance failed because some error occurred

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

uint32_t flags

Flags that specify extra capability for mac driver

struct eth_spi_custom_driver_config_t

Custom SPI Driver Configuration. This structure declares configuration and callback functions to access Ethernet SPI module via user’s custom SPI driver.

Public Members

void *config

Custom driver specific configuration data used by init() function.

Note: Type and its content is fully under user’s control

void *(*init)(const void *spi_config)

Custom driver SPI Initialization.

Note: return type and its content is fully under user’s control

Param spi_config [in] Custom driver specific configuration

Return

- spi_ctx: when initialization is successful, a pointer to context structure holding all variables needed for subsequent SPI access operations (e.g. SPI bus identification, mutexes, etc.)
- NULL: driver initialization failed
**esp_err_t (deinit)**(void *spi_ctx)

Custom driver De-initialization.

**Param spi_ctx [in]** a pointer to driver specific context structure

**Return**

- ESP_OK: driver de-initialization was successful
- ESP_FAIL: driver de-initialization failed
- any other failure codes are allowed to be used to provide failure isolation

**esp_err_t (read)**(void *spi_ctx, uint32_t cmd, uint32_t addr, void *data, uint32_t data_len)

Custom driver SPI read.

**Note:** The read function is responsible to construct command, address and data fields of the SPI frame in format expected by particular SPI Ethernet module

**Param spi_ctx [in]** a pointer to driver specific context structure

**Param cmd [in]** command

**Param addr [in]** register address

**Param data [out]** read data

**Param data_len [in]** read data length in bytes

**Return**

- ESP_OK: read was successful
- ESP_FAIL: read failed
- any other failure codes are allowed to be used to provide failure isolation

**esp_err_t (write)**(void *spi_ctx, uint32_t cmd, uint32_t addr, const void *data, uint32_t data_len)

Custom driver SPI write.

**Note:** The write function is responsible to construct command, address and data fields of the SPI frame in format expected by particular SPI Ethernet module

**Param spi_ctx [in]** a pointer to driver specific context structure

**Param cmd [in]** command

**Param addr [in]** register address

**Param data [in]** data to write

**Param data_len [in]** length of data to write in bytes

**Return**

- ESP_OK: write was successful
- ESP_FAIL: write failed
- any other failure codes are allowed to be used to provide failure isolation

**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.
Chapter 2. API Reference

**ETH_DEFAULT_SPI**

Default configuration of the custom SPI driver. Internal ESP-IDF SPI Master driver is used by default.

**Type Definitions**

typedef struct esp_eth_mac_s esp_eth_mac_t

Ethernet MAC.

**Enumerations**

enum emac_rmii_clock_mode_t

RMII Clock Mode Options.

Values:

enumerator EMAC_CLK_DEFAULT

Default values configured using Kconfig are going to be used when “Default” selected.

enumerator EMAC_CLK_EXT_IN

Input RMII Clock from external. EMAC Clock GPIO number needs to be configured when this option is selected.

**Note:** MAC will get RMII clock from outside. Note that ESP32 only supports GPIO0 to input the RMII clock.

enumerator EMAC_CLK_OUT

Output RMII Clock from internal APLL Clock. EMAC Clock GPIO number needs to be configured when this option is selected.

enum emac_rmii_clock_gpio_t

RMII Clock GPIO number Options.

Values:

enumerator EMAC_CLK_IN_GPIO

MAC will get RMII clock from outside at this GPIO.

**Note:** ESP32 only supports GPIO0 to input the RMII clock.

enumerator EMAC_APPL_CLK_OUT_GPIO

Output RMII Clock from internal APLL Clock available at GPIO0.

**Note:** 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.
enumerator **EMAC_CLK_OUT_GPIO**
Output RMII Clock from internal APLL Clock available at GPIO16.

enumerator **EMAC_CLK_OUT_180_GPIO**
Inverted Output RMII Clock from internal APLL Clock available at GPIO17.

**Header File**

- components/esp_eth/include/esp_eth_phy.h
- This header file can be included with:

```c
#include "esp_eth_phy.h"
```

- This header file is a part of the API provided by the esp_eth component. To declare that your component depends on esp_eth, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_eth
```

or

```cmake
PRIV_REQUIRES esp_eth
```

**Functions**

```c
esp_eth_phy_t *esp_eth_phy_new_ip101 (const eth_phy_config_t *config)
```

Create a PHY instance of IP101.

**Parameters config** — [in] configuration of PHY

**Returns**

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

```c
esp_eth_phy_t *esp_eth_phy_new_rtl8201 (const eth_phy_config_t *config)
```

Create a PHY instance of RTL8201.

**Parameters config** — [in] configuration of PHY

**Returns**

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

```c
esp_eth_phy_t *esp_eth_phy_new_lan87xx (const eth_phy_config_t *config)
```

Create a PHY instance of LAN87xx.

**Parameters config** — [in] configuration of PHY

**Returns**

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

```c
esp_eth_phy_t *esp_eth_phy_new_dp83848 (const eth_phy_config_t *config)
```

Create a PHY instance of DP83848.

**Parameters config** — [in] configuration of PHY

**Returns**

- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

```c
esp_eth_phy_t *esp_eth_phy_new_ksz80xx (const eth_phy_config_t *config)
```

Create a PHY instance of KSZ80xx.

The phy model from the KSZ80xx series is detected automatically. If the driver is unable to detect a supported model, NULL is returned.
Currently, the following models are supported: KSZ8001, KSZ8021, KSZ8031, KSZ8041, KSZ8051, KSZ8061, KSZ8081, KSZ8091

**Parameters config** [in] configuration of PHY

**Returns**
- instance: create PHY instance successfully
- NULL: create PHY instance failed because some error occurred

**Structures**

struct esp_eth_phy_s

Ethernet PHY.

**Public Members**

```c
esp_err_t (*set_mediator)(esp_eth_phy_t *phy, esp_eth_mediator_t *mediator)
```
Set mediator for PHY.

- **Param phy** [in] Ethernet PHY instance
- **Param mediator** [in] mediator of Ethernet driver

**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

```c
esp_err_t (*reset)(esp_eth_phy_t *phy)
```
Software Reset Ethernet PHY.

- **Param phy** [in] Ethernet PHY instance

**Return**
- ESP_OK: reset Ethernet PHY successfully
- ESP_FAIL: reset Ethernet PHY failed because some error occurred

```c
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

```c
esp_err_t (*init)(esp_eth_phy_t *phy)
```
Initialize Ethernet PHY.

- **Param phy** [in] Ethernet PHY instance

**Return**
- ESP_OK: initialize Ethernet PHY successfully
- ESP_FAIL: initialize Ethernet PHY failed because some error occurred

```c
esp_err_t (*deinit)(esp_eth_phy_t *phy)
```
Deinitialize Ethernet PHY.
**Param phy** [in] Ethernet PHY instance
**Return**
- ESP_OK: deinitialize Ethernet PHY successfully
- ESP_FAIL: deinitialize Ethernet PHY failed because some error occurred

```
esp_err_t (*autonego_ctrl)(esp_eth_phy_t *phy, eth_phy_autoneg_cmd_t cmd, bool *autonego_en_stat)
```
Configure auto negotiation.

**Param phy** [in] Ethernet PHY instance
**Param cmd** [in] Configuration command, it is possible to Enable (restart), Disable or get current status of PHY autonegotiation
**Param autonego_en_stat** [out] Address where to store current status of auto negotiation configuration

**Return**
- ESP_OK: restart auto negotiation successfully
- ESP_FAIL: restart auto negotiation failed because some error occurred
- ESP_ERR_INVALID_ARG: invalid command

```
esp_err_t (*get_link)(esp_eth_phy_t *phy)
```
Get Ethernet PHY link status.

**Param phy** [in] Ethernet PHY instance
**Return**
- ESP_OK: get Ethernet PHY link status successfully
- ESP_FAIL: get Ethernet PHY link status failed because some error occurred

```
esp_err_t (*pwrctl)(esp_eth_phy_t *phy, bool enable)
```
Power control of Ethernet PHY.

**Param phy** [in] Ethernet PHY instance
**Param enable** [in] set true to power on Ethernet PHY; set false to power off Ethernet PHY

**Return**
- ESP_OK: control Ethernet PHY power successfully
- ESP_FAIL: control Ethernet PHY power failed because some error occurred

```
esp_err_t (*set_addr)(esp_eth_phy_t *phy, uint32_t addr)
```
Set PHY chip address.

**Param phy** [in] Ethernet PHY instance
**Param addr** [in] PHY chip address

**Return**
- ESP_OK: set Ethernet PHY address successfully
- ESP_FAIL: set Ethernet PHY address failed because some error occurred

```
esp_err_t (*get_addr)(esp_eth_phy_t *phy, uint32_t *addr)
```
Get PHY chip address.

**Param phy** [in] Ethernet PHY instance
**Param addr** [out] PHY chip address

**Return**
- ESP_OK: get Ethernet PHY address successfully
- ESP_FAIL: get Ethernet PHY address failed because of invalid argument
- ESP_ERR_INVALID_ARG: get Ethernet PHY address failed because of invalid argument

```
esp_err_t (*advertise_pause_ability)(esp_eth_phy_t *phy, uint32_t ability)
```
Advertise pause function supported by MAC layer.

**Param phy** [in] Ethernet PHY instance
**Param addr [out] Pause ability**

**Return**
- ESP_OK: Advertise pause ability successfully
- ESP_ERR_INVALID_ARG: Advertise pause ability failed because of invalid argument

`esp_err_t (*loopback)(esp_eth_phy_t *phy, bool enable)`

Sets the PHY to loopback mode.

**Param phy [in] Ethernet PHY instance**
**Param enable [in] enables or disables PHY loopback**

**Return**
- ESP_OK: PHY instance loopback mode has been configured successfully
- ESP_FAIL: PHY instance loopback configuration failed because some error occurred

`esp_err_t (*set_speed)(esp_eth_phy_t *phy, eth_speed_t speed)`

Sets PHY speed mode.

**Note:** Autonegotiation feature needs to be disabled prior to calling this function for the new setting to be applied

**Param phy [in] Ethernet PHY instance**
**Param speed [in] Speed mode to be set**

**Return**
- ESP_OK: PHY instance speed mode has been configured successfully
- ESP_FAIL: PHY instance speed mode configuration failed because some error occurred

`esp_err_t (*set_duplex)(esp_eth_phy_t *phy, eth_duplex_t duplex)`

Sets PHY duplex mode.

**Note:** Autonegotiation feature needs to be disabled prior to calling this function for the new setting to be applied

**Param phy [in] Ethernet PHY instance**
**Param duplex [in] Duplex mode to be set**

**Return**
- ESP_OK: PHY instance duplex mode has been configured successfully
- ESP_FAIL: PHY instance duplex mode configuration failed because some error occurred

`esp_err_t (*custom_ioctl)(esp_eth_phy_t *phy, uint32_t cmd, void *data)`

Custom IO function of PHY driver. This function is intended to extend common options of esp_eth_ioctl to cover specifics of PHY chip.

**Note:** This function may not be assigned when the PHY chip supports only most common set of configuration options.
Chapter 2. API Reference

- 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
- ESP_ERR_NOT_SUPPORTED: requested feature is not supported

```c
esp_err_t (*del)(esp_eth_phy_t *phy)
```
Free memory of Ethernet PHY instance.

**Param phy** [in] Ethernet PHY instance

**Return**
- ESP_OK: free PHY instance successfully
- ESP_FAIL: free PHY instance failed because some error occurred

```c
struct eth_phy_config_t
```
Ethernet PHY configuration.

**Public Members**

```c
int32_t phy_addr
```
PHY address, set -1 to enable PHY address detection at initialization stage

```c
uint32_t reset_timeout_ms
```
Reset timeout value (Unit: ms)

```c
uint32_t autonego_timeout_ms
```
Auto-negotiation timeout value (Unit: ms)

```c
int reset_gpio_num
```
Reset GPIO number, -1 means no hardware reset

**Macros**

```c
ESP_ETH_PHY_ADDR_AUTO
ETH_PHY_DEFAULT_CONFIG()
```
Default configuration for Ethernet PHY object.

**Type Definitions**

```c
typedef struct esp_eth_phy_s esp_eth_phy_t
```
Ethernet PHY.

**Enumerations**

```c
enum eth_phy_autoneg_cmd_t
```
Auto-negotiation control commands.

**Values**:

```c
enumerator ESP_ETH_PHY_AUTONEGO_RESTART
```
enumerator **ESP_ETH_PHY_AUTONEGO_EN**

enumerator **ESP_ETH_PHY_AUTONEGO_DIS**

enumerator **ESP_ETH_PHY_AUTONEGO_G_STAT**

**Header File**

- components/esp/include/esp_eth_phy_802_3.h
- This header file can be included with:

```
#include "esp_eth_phy_802_3.h"
```

- This header file is a part of the API provided by the `esp_eth` component. To declare that your component depends on `esp_eth`, add the following to your CMakeLists.txt:

```
REQUIRES esp_eth
```

or

```
PRIV_REQUIRES esp_eth
```

**Functions**

`esp_err_t esp_eth_phy_802_3_set_mediator(phy_802_3_t *phy_802_3, esp_eth_mediator_t *eth)`

Set Ethernet mediator.

**Parameters**

- `phy_802_3` – IEEE 802.3 PHY object infostructure
- `eth` – Ethernet mediator pointer

**Returns**

- ESP_OK: Ethernet mediator set successfully
- ESP_ERR_INVALID_ARG: if `eth` is NULL

`esp_err_t esp_eth_phy_802_3_reset(phy_802_3_t *phy_802_3)`

Reset PHY.

**Parameters**

- `phy_802_3` – IEEE 802.3 PHY object infostructure

**Returns**

- ESP_OK: Ethernet PHY reset successfully
- ESP_FAIL: reset Ethernet PHY failed because some error occurred

`esp_err_t esp_eth_phy_802_3_autonego_ctrl(phy_802_3_t *phy_802_3, eth_phy_autoneg_cmd_t cmd, bool *autonego_en_stat)`

Control autonegotiation mode of Ethernet PHY.

**Parameters**

- `phy_802_3` – IEEE 802.3 PHY object infostructure
- `cmd` – autonegotiation command enumeration
- `autonego_en_stat` – [out] autonegotiation enabled flag

**Returns**

- ESP_OK: Ethernet PHY autonegotiation configured successfully
- ESP_FAIL: Ethernet PHY autonegotiation configuration fail because some error occured
- ESP_ERR_INVALID_ARG: invalid value of `cmd`

`esp_err_t esp_eth_phy_802_3_pwrctl(phy_802_3_t *phy_802_3, bool enable)`

Power control of Ethernet PHY.

**Parameters**

- `phy_802_3` – IEEE 802.3 PHY object infostructure
Enable Ethernet PHY power mode.

Returns
- ESP_OK: Ethernet PHY power down mode set successfully
- ESP_FAIL: Ethernet PHY power up or power down failed because some error occurred

```c
esp_err_t esp_eth_phy_802_3_set_power(uint8_t enable)
```

Set Ethernet PHY address.

Parameters
- **phy_802_3** – IEEE 802.3 PHY object infostructure
- **addr** – new PHY address

Returns
- ESP_OK: Ethernet PHY address set

```c
esp_err_t esp_eth_phy_802_3_set_addr(phy_802_3_t *phy_802_3, uint32_t addr)
```

Get Ethernet PHY address.

Parameters
- **phy_802_3** – IEEE 802.3 PHY object infostructure
- **addr** – [out] Ethernet PHY address

Returns
- ESP_OK: Ethernet PHY address read successfully
- ESP_ERR_INVALID_ARG: addr pointer is NULL

```c
esp_err_t esp_eth_phy_802_3_get_addr(phy_802_3_t *phy_802_3, uint32_t *addr)
```

Advertise pause function ability.

Parameters
- **phy_802_3** – IEEE 802.3 PHY object infostructure
- **ability** – enable or disable pause ability

Returns
- ESP_OK: pause ability set successfully
- ESP_FAIL: Advertise pause function ability failed because some error occurred

```c
esp_err_t esp_eth_phy_802_3_advertise_pause_ability(phy_802_3_t *phy_802_3, uint32_t ability)
```

Set Ethernet PHY loopback mode.

Parameters
- **phy_802_3** – IEEE 802.3 PHY object infostructure
- **enable** – set true to enable loopback; set false to disable loopback

Returns
- ESP_OK: Ethernet PHY loopback mode set successfully
- ESP_FAIL: Ethernet PHY loopback configuration failed because some error occurred

```c
esp_err_t esp_eth_phy_802_3_loopback(phy_802_3_t *phy_802_3, bool enable)
```

Set Ethernet PHY speed.

Parameters
- **phy_802_3** – IEEE 802.3 PHY object infostructure
- **speed** – new speed of Ethernet PHY link

Returns
- ESP_OK: Ethernet PHY speed set successfully
- ESP_FAIL: Set Ethernet PHY speed failed because some error occurred

```c
esp_err_t esp_eth_phy_802_3_set_speed(phy_802_3_t *phy_802_3, eth_speed_t speed)
```

Set Ethernet PHY duplex mode.

Parameters
- **phy_802_3** – IEEE 802.3 PHY object infostructure
- **duplex** – new duplex mode for Ethernet PHY link

Returns
- ESP_OK: Ethernet PHY duplex mode set successfully

```c
esp_err_t esp_eth_phy_802_3_set_duplex(phy_802_3_t *phy_802_3, eth_duplex_t duplex)
```
Chapter 2. API Reference

- ESP_ERR_INVALID_STATE: unable to set duplex mode to Half if loopback is enabled
- ESP_FAIL: Set Ethernet PHY duplex mode failed because some error occurred

`esp_err_t esp_eth_phy_802_3_init (phy_802_3_t *phy_802_3)`
Initialize Ethernet PHY.

Parameters
- `phy_802_3` - IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethernet PHY initialized successfully

`esp_err_t esp_eth_phy_802_3_deinit (phy_802_3_t *phy_802_3)`
Power off Ethernet PHY.

Parameters
- `phy_802_3` - IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethernet PHY powered off successfully

`esp_err_t esp_eth_phy_802_3_del (phy_802_3_t *phy_802_3)`
Delete Ethernet PHY infostructure.

Parameters
- `phy_802_3` - IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: Ethernet PHY infostructure deleted

`esp_err_t esp_eth_phy_802_3_reset_hw (phy_802_3_t *phy_802_3, uint32_t reset_assert_us)`
Performs hardware reset with specific reset pin assertion time.

Parameters
- `phy_802_3` - IEEE 802.3 PHY object infostructure
- `reset_assert_us` - Hardware reset pin assertion time

Returns
- ESP_OK: reset Ethernet PHY successfully

`esp_err_t esp_eth_phy_802_3_detect_phy_addr (esp_eth_mediator_t *eth, int *detected_addr)`
Detect PHY address.

Parameters
- `eth` - Mediator of Ethernet driver
- `detected_addr` - [out] a valid address after detection

Returns
- 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

`esp_err_t esp_eth_phy_802_3_basic_phy_init (phy_802_3_t *phy_802_3)`
Performs basic PHY chip initialization.

**Note:** It should be called as the first function in PHY specific driver instance

Parameters
- `phy_802_3` - IEEE 802.3 PHY object infostructure

Returns
- ESP_OK: initialized Ethernet PHY successfully
- ESP_FAIL: initialization of Ethernet PHY failed because some error occurred
- ESP_ERR_INVALID_ARG: invalid argument
- ESP_ERR_NOT_FOUND: PHY device not detected
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation
**esp_err_t esp_eth_phy_802_3_basic_phy_deinit (phy_802_3_t *phy_802_3)**

Performs basic PHY chip de-initialization.

**Note:** It should be called as the last function in PHY specific driver instance

**Parameters**
- `phy_802_3` - IEEE 802.3 PHY object infostructure

**Returns**
- ESP_OK: de-initialized Ethernet PHY successfully
- ESP_FAIL: de-initialization of Ethernet PHY failed because some error occurred
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

**esp_err_t esp_eth_phy_802_3_read_oui (phy_802_3_t *phy_802_3, uint32_t *oui)**

Reads raw content of OUI field.

**Parameters**
- `phy_802_3` - IEEE 802.3 PHY object infostructure
- `oui` - [out] OUI value

**Returns**
- ESP_OK: OUI field read successfully
- ESP_FAIL: OUI field read failed because some error occurred
- ESP_ERR_INVALID_ARG: invalid oui argument
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

**esp_err_t esp_eth_phy_802_3_read_manufac_info (phy_802_3_t *phy_802_3, uint8_t *model, uint8_t *rev)**

Reads manufacturer’s model and revision number.

**Parameters**
- `phy_802_3` - IEEE 802.3 PHY object infostructure
- `model` - [out] Manufacturer’s model number (can be NULL when not required)
- `rev` - [out] Manufacturer’s revision number (can be NULL when not required)

**Returns**
- ESP_OK: Manufacturer info read successfully
- ESP_FAIL: Manufacturer info read failed because some error occurred
- ESP_ERR_TIMEOUT: MII Management read/write operation timeout
- ESP_ERR_INVALID_STATE: PHY is in invalid state to perform requested operation

**inline phy_802_3_t *esp_eth_phy_into_phy_802_3 (esp_eth_phy_t *phy)**

Returns address to parent IEEE 802.3 PHY object infostructure.

**Parameters**
- `phy` - Ethernet PHY instance

**Returns**
- `phy_802_3_t` - address to parent IEEE 802.3 PHY object infostructure

**esp_err_t esp_eth_phy_802_3_obj_config_init (phy_802_3_t *phy_802_3, const eth_phy_config_t *config)**

Initializes configuration of parent IEEE 802.3 PHY object infostructure.

**Parameters**
- `phy_802_3` - Address to IEEE 802.3 PHY object infostructure
- `config` - Configuration of the IEEE 802.3 PHY object

**Returns**
- ESP_OK: configuration initialized successfully
- ESP_ERR_INVALID_ARG: invalid config argument
Chapter 2. API Reference

Structures

```c
struct phy_802_3_t
    IEEE 802.3 PHY object info structure.
```

Public Members

```c
    esp_eth_phy_t parent
    Parent Ethernet PHY instance
```

```c
    esp_eth_mediator_t *eth
    Mediator of Ethernet driver
```

```c
    int addr
    PHY address
```

```c
    uint32_t reset_timeout_ms
    Reset timeout value (Unit: ms)
```

```c
    uint32_t autonego_timeout_ms
    Auto-negotiation timeout value (Unit: ms)
```

```c
    eth_link_t link_status
    Current Link status
```

```c
    int reset_gpio_num
    Reset GPIO number, -1 means no hardware reset
```

Header File

- components/esp_eth/include/esp_eth_netif_glue.h
- This header file can be included with:
  ```c
  #include "esp_eth_netif_glue.h"
  ```
- This header file is a part of the API provided by the esp_eth component. To declare that your component depends on esp_eth, add the following to your CMakeLists.txt:
  ```
  REQUIRES esp_eth
  or
  PRIV_REQUIRES esp_eth
  ```

Functions

```c
    esp_eth_netif_glue_handle_t 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

**Parameters**

- `eth_hdl` - Ethernet driver handle

**Returns**

- `glue` object, which inherits esp_netif_driver_base_t
Chapter 2. API Reference

```c
esp_err_t esp_eth_del_netif_glue(esp_eth_netif_glue_handle_t eth_netif_glue)
```

Delete netif glue of Ethernet driver.

- **Parameters** `eth_netif_glue` - netif glue
- **Returns** -ESP_OK: delete netif glue successfully

### Type Definitions

typedef struct esp_eth_netif_glue_t *esp_eth_netif_glue_handle_t

Handle of netif glue - an intermediate layer between netif and Ethernet driver.

Code examples for the Ethernet API are provided in the `ethernet` directory of ESP-IDF examples.

#### 2.4.3 Thread

**Thread**

Introduction  Thread is an IP-based mesh networking protocol. It is based on the 802.15.4 physical and MAC layer.

Application Examples  The openthread directory of ESP-IDF examples contains the following applications:

- The OpenThread interactive shell `openthread/ot_cli`
- The Thread Border Router `openthread/ot_br`
- The Thread Radio Co-Processor `openthread/ot_rcp`

API Reference  For manipulating the Thread network, the OpenThread API shall be used. The OpenThread API docs can be found at the [OpenThread API docs](#).

ESP-IDF provides extra APIs for launching and managing the OpenThread stack, binding to network interfaces and border routing features.

**Header File**

- `components/openthread/include/esp_openthread.h`
- This header file can be included with:

```c
#include "esp_openthread.h"
```

- This header file is a part of the API provided by the `openthread` component. To declare that your component depends on `openthread`, add the following to your `CMakeLists.txt`:

```c
REQUIRES openthread
```

or

```c
PRIV_REQUIRES openthread
```

**Functions**

```c
esp_err_t esp_openthread_init(const esp_openthread_platform_config_t *init_config)
```

Initializes the full OpenThread stack.

- **Note:** The OpenThread instance will also be initialized in this function.
Chapter 2. API Reference

Parameters `init_config` [in] The initialization configuration.

Returns
- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_ERR_INVALID_ARG if radio or host connection mode not supported
- ESP_ERR_INVALID_STATE if already initialized

`esp_err_t esp_openthread_auto_start (otOperationalDatasetTlvs *datasetTlvs)`

Starts the Thread protocol operation and attaches to a Thread network.

Parameters `datasetTlvs` [in] The operational dataset (TLV encoded), if it’s NULL, the function will generate the dataset based on the configurations from kconfig.

Returns
- ESP_OK on success
- ESP_FAIL on failures

`esp_err_t esp_openthread_launch_mainloop (void)`

Launches the OpenThread main loop.

**Note:** This function will not return unless error happens when running the OpenThread stack.

Returns
- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_FAIL on other failures

`esp_err_t esp_openthread_deinit (void)`

This function performs OpenThread stack and platform driver deinitialization.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized

`otInstance *esp_openthread_get_instance (void)`

This function acquires the underlying OpenThread instance.

**Note:** This function can be called on other tasks without lock.

Returns The OpenThread instance pointer

Header File

- `components/openthread/include/esp_openthread_types.h`
- This header file can be included with:
  ```
  #include "esp_openthread_types.h"
  ```
- This header file is a part of the API provided by the `openthread` component. To declare that your component depends on `openthread`, add the following to your CMakeLists.txt:
  ```
  REQUIRE openthread
  ```
  or
  ```
  PRIV_REQUIRE openthread
  ```
Structures

struct esp_openthread_role_changed_event_t
    OpenThread role changed event data.

    Public Members

    otDeviceRole previous_role
        Previous Thread role

    otDeviceRole current_role
        Current Thread role

struct esp_openthread_mainloop_context_t
    This structure represents a context for a select() based mainloop.

    Public Members

    fd_set read_fds
        The read file descriptors

    fd_set write_fds
        The write file descriptors

    fd_set error_fds
        The error file descriptors

    int max_fd
        The max file descriptor

    struct timeval timeout
        The timeout

struct esp_openthread_uart_config_t
    The uart port config for OpenThread.

    Public Members

    uart_port_t port
        UART port number

    uart_config_t uart_config
        UART configuration, see uart_config_t docs

    gpio_num_t rx_pin
        UART RX pin
### Chapter 2. API Reference

#### gpio_num_t tx_pin
UART TX pin

```c
struct esp_openthread_spi_host_config_t
```

The spi port config for OpenThread.

**Public Members**

- `spi_host_device_t host_device`
  - SPI host device

- `spi_dma_chan_t dma_channel`
  - DMA channel

- `spi_bus_config_t spi_interface`
  - SPI bus

- `spi_device_interface_config_t spi_device`
  - SPI peripheral device

- `gpio_num_t intr_pin`
  - SPI interrupt pin

```c
struct esp_openthread_spi_slave_config_t
```

The spi slave config for OpenThread.

**Public Members**

- `spi_host_device_t host_device`
  - SPI host device

- `spi_bus_config_t bus_config`
  - SPI bus config

- `spi_slave_interface_config_t slave_config`
  - SPI slave config

- `gpio_num_t intr_pin`
  - SPI interrupt pin

```c
struct esp_openthread_radio_config_t
```

The OpenThread radio configuration.

**Public Members**
Chapter 2. API Reference

```c
esp_openthread_radio_mode_t radio_mode
    The radio mode

esp_openthread_uart_config_t radio_uart_config
    The uart configuration to RCP

esp_openthread_spi_host_config_t radio_spi_config
    The spi configuration to RCP
```

```c
struct esp_openthread_host_connection_config_t
    The OpenThread host connection configuration.
```

**Public Members**

```c
esp_openthread_host_connection_mode_t host_connection_mode
    The host connection mode

esp_openthread_uart_config_t host_uart_config
    The uart configuration to host

usb_serial_jtag_driver_config_t host_usb_config
    The usb configuration to host

esp_openthread_spi_slave_config_t spi_slave_config
    The spi configuration to host
```

```c
struct esp_openthread_port_config_t
    The OpenThread port specific configuration.
```

**Public Members**

```c
const char *storage_partition_name
    The partition for storing OpenThread dataset

uint8_t netif_queue_size
    The packet queue size for the network interface

uint8_t task_queue_size
    The task queue size
```

```c
struct esp_openthread_platform_config_t
    The OpenThread platform configuration.
```

**Public Members**
**Chapter 2. API Reference**

```c
typedef void (*esp_openthread_rcp_failure_handler)(void)
```

**Type Definitions**

```c
#define OPENTHREAD_EVENT_START
#define OPENTHREAD_EVENT_STOP
#define OPENTHREAD_EVENT_DETACHED
#define OPENTHREAD_EVENT_ATTACHED
#define OPENTHREAD_EVENT_ROLE_CHANGED
#define OPENTHREAD_EVENT_IF_UP
#define OPENTHREAD_EVENT_IF_DOWN
#define OPENTHREAD_EVENT_GOT_IP6
#define OPENTHREAD_EVENT_LOST_IP6
#define OPENTHREAD_EVENT_MULTICAST_GROUP_JOIN
```

**Enumerations**

```c
typedef enum esp_openthread_event_t
{
    OPENTHREAD_EVENT_START,
    OPENTHREAD_EVENT_STOP,
    OPENTHREAD_EVENT_DETACHED,
    OPENTHREAD_EVENT_ATTACHED,
    OPENTHREAD_EVENT_ROLE_CHANGED,
    OPENTHREAD_EVENT_IF_UP,
    OPENTHREAD_EVENT_IF_DOWN,
    OPENTHREAD_EVENT_GOT_IP6,
    OPENTHREAD_EVENT_LOST_IP6,
    OPENTHREAD_EVENT_MULTICAST_GROUP_JOIN
} esp_openthread_event_t;
```

**Values:**

- `OPENTHREAD_EVENT_START`: OpenThread stack start
- `OPENTHREAD_EVENT_STOP`: OpenThread stack stop
- `OPENTHREAD_EVENT_DETACHED`: OpenThread detached
- `OPENTHREAD_EVENT_ATTACHED`: OpenThread attached
- `OPENTHREAD_EVENT_ROLE_CHANGED`: OpenThread role changed
- `OPENTHREAD_EVENT_IF_UP`: OpenThread network interface up
- `OPENTHREAD_EVENT_IF_DOWN`: OpenThread network interface down
- `OPENTHREAD_EVENT_GOT_IP6`: OpenThread stack added IPv6 address
- `OPENTHREAD_EVENT_LOST_IP6`: OpenThread stack removed IPv6 address
- `OPENTHREAD_EVENT_MULTICAST_GROUP_JOIN`: OpenThread stack joined IPv6 multicast group
enumerator **OPENTHREAD_EVENT_MULTICAST_GROUP_LEAVE**  
OpenThread stack left IPv6 multicast group

enumerator **OPENTHREAD_EVENT_TREL_ADD_IP6**  
OpenThread stack added TREL IPv6 address

enumerator **OPENTHREAD_EVENT_TREL_REMOVE_IP6**  
OpenThread stack removed TREL IPv6 address

enumerator **OPENTHREAD_EVENT_TREL_MULTICAST_GROUP_JOIN**  
OpenThread stack joined TREL IPv6 multicast group

enumerator **OPENTHREAD_EVENT_SET_DNS_SERVER**  
OpenThread stack set DNS server

**enum esp_openthread_radio_mode_t**

The radio mode of OpenThread.

*Values:*

enumerator **RADIO_MODE_NATIVE**  
Use the native 15.4 radio

enumerator **RADIO_MODE_UART_RCP**  
UART connection to a 15.4 capable radio co-processor (RCP)

enumerator **RADIO_MODE_SPI_RCP**  
SPI connection to a 15.4 capable radio co-processor (RCP)

enumerator **RADIO_MODE_MAX**  
Using for parameter check

**enum esp_openthread_host_connection_mode_t**

How OpenThread connects to the host.

*Values:*

enumerator **HOST_CONNECTION_MODE_NONE**  
Disable host connection

enumerator **HOST_CONNECTION_MODE_CLI_UART**  
CLI UART connection to the host

enumerator **HOST_CONNECTION_MODE_CLI_USB**  
CLI USB connection to the host

enumerator **HOST_CONNECTION_MODE_RCP_UART**  
RCP UART connection to the host

enumerator **HOST_CONNECTION_MODE_RCP_SPI**  
RCP SPI connection to the host
enumerator HOST_CONNECTION_MODE_MAX

Using for parameter check

Header File

- components/openthread/include/esp_openthread_lock.h
- This header file can be included with:
  ```
  #include "esp_openthread_lock.h"
  ```
- This header file is a part of the API provided by the openthread component. To declare that your component depends on openthread, add the following to your CMakeLists.txt:
  ```
  REQUIRES openthread
  ```
  or
  ```
  PRIV_REQUIRES openthread
  ```

Functions

**esp_err_t esp_openthread_lock_init (void)**

This function initializes the OpenThread API lock.

**Returns**

- ESP_OK on success
- ESP_ERR_NO_MEM if allocation has failed
- ESP_ERR_INVALID_STATE if already initialized

**void esp_openthread_lock_deinit (void)**

This function deinitializes the OpenThread API lock.

**bool esp_openthread_lock_acquire (TickType_t block_ticks)**

This function acquires the OpenThread API lock.

**Parameters** block_ticks -[in] The maximum number of RTOS ticks to wait for the lock.

**Returns**

- True on lock acquired
- False on failing to acquire the lock with the timeout.

**void esp_openthread_lock_release (void)**

This function releases the OpenThread API lock.

**bool esp_openthread_task_switching_lock_acquire (TickType_t block_ticks)**

This function acquires the OpenThread API task switching lock.

**Note:** Every OT APIs that takes an otInstance argument MUST be protected with this API lock except that the call site is in OT callbacks.

**Parameters** block_ticks -[in] The maximum number of RTOS ticks to wait for the lock.

**Note:** In OpenThread API context, it waits for some actions to be done in other tasks (like lwip), after task switching, it needs to call OpenThread API again. Normally it’s not allowed, since the previous OpenThread API lock is not released yet. This task_switching lock allows the OpenThread API can be called in this case.

**Note:** Please use esp_openthread_lock_acquire() for normal cases.
Returns

- True on lock acquired
- False on failing to acquire the lock with the timeout.

void esp_openthread_task_switching_lock_release (void)

This function releases the OpenThread API task switching lock.

Header File

- components/openthread/include/esp_openthread_netif_glue.h
  - This header file can be included with:
    ```
    #include "esp_openthread_netif_glue.h"
    ```
  - This header file is a part of the API provided by the openthread component. To declare that your component depends on openthread, add the following to your CMakeLists.txt:
    ```
    REQUIRES openthread
    or
    PRIV_REQUIRES openthread
    ```

Functions

void *esp_openthread_netif_glue_init (const esp_openthread_platform_config_t *config)

This function initializes the OpenThread network interface glue.

Parameters

config - [in] The platform configuration.

Returns

- glue pointer on success
- NULL on failure

void esp_openthread_netif_glue_deinit (void)

This function deinitializes the OpenThread network interface glue.

esp_netif_t *esp_openthread_get_netif (void)

This function acquires the OpenThread netif.

Returns

- The OpenThread netif or NULL if not initialized.

Macros

ESP_NETIF_INHERENT_DEFAULT_OPENTHREAD ()

Default configuration reference of OT esp-netif.

ESP_NETIF_DEFAULT_OPENTHREAD ()

Header File

- components/openthread/include/esp_openthread_border_router.h
  - This header file can be included with:
    ```
    #include "esp_openthread_border_router.h"
    ```
  - This header file is a part of the API provided by the openthread component. To declare that your component depends on openthread, add the following to your CMakeLists.txt:
    ```
    REQUIRES openthread
    or
    ```
Functions

void esp_openthread_set_backbone_netif (esp_netif_t *backbone_netif)
Sets the backbone interface used for border routing.

Note: This function must be called before esp_openthread_init

Parameters
backbone_netif – [in] The backbone network interface (WiFi or ethernet)

esp_err_t esp_openthread_border_router_init (void)
Initializes the border router features of OpenThread.

Note: Calling this function will make the device behave as an OpenThread border router. Kconfig option CONFIG_OPENTHREAD_BORDER_ROUTER is required.

Returns
• ESP_OK on success
• ESP_ERR_NOT_SUPPORTED if feature not supported
• ESP_ERR_INVALID_STATE if already initialized
• ESP_FAIL on other failures

esp_err_t esp_openthread_border_router_deinit (void)
Deinitializes the border router features of OpenThread.

Returns
• ESP_OK on success
• ESP_ERR_INVALID_STATE if not initialized
• ESP_FAIL on other failures

esp_netif_t *esp_openthread_get_backbone_netif (void)
Gets the backbone interface of OpenThread border router.

Returns
The backbone interface or NULL if border router not initialized.

void esp_openthread_register_rcp_failure_handler (esp_openthread_rcp_failure_handler handler)
Registers the callback for RCP failure.

void esp_openthread_rcp_deinit (void)
Deinitializes the connection to RCP.

Thread is an IPv6-based mesh networking technology for IoT.

Code examples for the Thread API are provided in the openthread directory of ESP-IDF examples.

2.4.4 ESP-NETIF

ESP-NETIF

The purpose of the ESP-NETIF library is twofold:
• It provides an abstraction layer for the application on top of the TCP/IP stack. This allows 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 allows different implementations.

It is also possible to use a custom TCP/IP stack with ESP-IDF, provided it implements BSD API. For more information on building ESP-IDF without lwIP, please refer to components/esp_netif_stack/README.md.

Some ESP-NETIF API functions are intended to be called by application code, for example, to get or set interface IP addresses, and 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 by the default network event handlers.

**ESP-NETIF Architecture**

```
| (A) USER CODE |
| | Apps |
| | init settings events |
| +-------------------------+ |
| | * | * |

| new/config get/set/apps | * | init |
| Apps (DHCP, SNTF) |
| | *
| | *** |
| **| DHCP |
| | *** |

**| **| NETIF |
| | **|
| | **|
| | **|
| | **|
| | **|
| **| **|
| | **|
| **| **|
| **| **|
| **| **|

| glue|----<----|---| esp_netif_transmit |----<------| netif_output |
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 
| | | | | | | | 

(continues on next 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 propagate to the driver, user code, and network stack
- User settings and runtime configuration

### ESP-NETIF Interaction

**A) User Code, Boilerplate**  
Overall application interaction with a specific IO driver for communication media and configured TCP/IP network stack is abstracted using ESP-NETIF APIs and is outlined as below:

1. **Initialization code**
   1. Initializes IO driver
   2. Creates a new instance of ESP-NETIF and configure it with
      - ESP-NETIF specific options (flags, behavior, 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 handler for customized behavior or new interfaces
      - Register handlers for app-related events (such as IP lost or acquired)

**B) Interaction with network interfaces using ESP-NETIF API**

1. Gets and sets TCP/IP-related parameters (DHCP, IP, etc)
2. Receives IP events (connect or disconnect)
3. Controls application lifecycle (set interface up or down)

**B) Communication Driver, IO Driver, and Media Driver**  
Communication driver plays these two important roles in relation to ESP-NETIF:

1. Event handlers: Defines behavior patterns of interaction with ESP-NETIF (e.g., ethernet link-up -> turn netif on)
2. Glue IO layer: Adapts the input or output functions to use ESP-NETIF transmit, receive, and free receive buffer
   - Installs driver_transmit to the appropriate ESP-NETIF object so that outgoing packets from the network stack are passed to the IO driver
   - Calls `esp_netif_receive()` to pass incoming data to the network stack
C) ESP-NETIF  ESP-NETIF serves as an intermediary between an IO driver and a network stack, connecting the packet data path between the two. It provides a set of interfaces for attaching a driver to an ESP-NETIF object at runtime and configures a network stack during compiling. Additionally, a set of APIs is provided to control the network interface lifecycle and its TCP/IP properties. As an overview, the ESP-NETIF public interface can be divided into six groups:

1) Initialization APIs (to create and configure ESP-NETIF instance)
2) Input or 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 API for basic network interface properties
5) Network stack abstraction API: enabling user interaction with TCP/IP stack
   • Set interface up or down
   • DHCP server and client API
   • DNS API
   • \textit{SNTP API}
6) Driver conversion utilities API

D) Network Stack  The network stack has no public interaction with application code with regard to public interfaces and shall be fully abstracted by ESP-NETIF API.

E) ESP-NETIF L2 TAP Interface  The ESP-NETIF L2 TAP interface is a mechanism in ESP-IDF used to access Data Link Layer (L2 per OSI/ISO) for frame reception and transmission from the user application. Its typical usage in the embedded world might be the implementation of non-IP-related protocols, e.g., PTP, Wake on LAN. Note that only Ethernet (IEEE 802.3) is currently supported.

From a user perspective, the ESP-NETIF L2 TAP interface is accessed using file descriptors of VFS, which provides file-like interfacing (using functions like \texttt{open()}, \texttt{read()}, \texttt{write()}, etc). To learn more, refer to \textit{Virtual Filesystem Component}.

There is only one ESP-NETIF L2 TAP interface device (path name) available. However multiple file descriptors with different configurations can be opened at a time since the ESP-NETIF L2 TAP interface can be understood as a generic entry point to the Layer 2 infrastructure. What is important is then the specific configuration of the particular file descriptor. It can be configured to give access to a specific Network Interface identified by \texttt{if_key} (e.g., \texttt{ETH_DEF}) and to filter only specific frames based on their type (e.g., Ethernet type in the case of IEEE 802.3). Filtering only specific frames is crucial since the ESP-NETIF L2 TAP needs to exist along with the IP stack and so the IP-related traffic (IP, ARP, etc.) should not be passed directly to the user application. Even though this option is still configurable, it is not recommended in standard use cases. Filtering is also advantageous from the perspective of the user’s application, as it only gets access to the frame types it is interested in, and the remaining traffic is either passed to other L2 TAP file descriptors or to the IP stack.

**ESP-NETIF L2 TAP Interface Usage Manual**

**Initialization**  To be able to use the ESP-NETIF L2 TAP interface, it needs to be enabled in Kconfig by \texttt{CONFIG_ESP_NETIF_L2_TAP} first and then registered by \texttt{esp_vfs_l2tap_intf_register()} prior usage of any VFS function.

\textbf{open()}  Once the ESP-NETIF L2 TAP is registered, it can be opened at path name \texttt{”/dev/net/tap”}. The same path name can be opened multiple times up to \texttt{CONFIG_ESP_NETIF_L2_TAP_MAX_FDS} and multiple file descriptors with a different configuration may access the Data Link Layer frames.

The ESP-NETIF L2 TAP can be opened with the \texttt{O_NONBLOCK} file status flag to make sure the \texttt{read()} does not block. Note that the \texttt{write()} may block in the current implementation when accessing a Network interface since
it is a shared resource among multiple ESP-NETIF L2 TAP file descriptors and IP stack, and there is currently no queuing mechanism deployed. The file status flag can be retrieved and modified using fcntl().

On success, open() returns the new file descriptor (a nonnegative integer). On error, -1 is returned, and errno is set to indicate the error.

ioctl() The newly opened ESP-NETIF L2 TAP file descriptor needs to be configured prior to its usage since it is not bounded to any specific Network Interface and no frame type filter is configured. The following configuration options are available to do so:

- **L2TAP_S_INTF_DEVICE** - bounds the file descriptor to a specific Network Interface that is identified by its if_key. ESP-NETIF Network Interface if_key is passed to ioctl() as the third parameter. Note that default Network Interfaces if_key’s used in ESP-IDF can be found in esp_netif/include/esp_netif_defaults.h.
- **L2TAP_S_DEVICE_DRV_HNDL** - is another way to bound the file descriptor to a specific Network Interface. In this case, the Network interface is identified directly by IO Driver handle (e.g., esp_eth_handle_t in case of Ethernet). The IO Driver handle is passed to ioctl() as the third parameter.
- **L2TAP_S_RCV_FILTER** - sets the filter to frames with the type to be passed to the file descriptor. In the case of Ethernet frames, the frames are to be filtered based on the Length and Ethernet type field. In case the filter value is set less than or equal to 0x05DC, the Ethernet type field is considered to represent IEEE802.3 Length Field, and all frames with values in interval <0, 0x05DC> at that field are passed to the file descriptor. The IEEE802.2 logical link control (LLC) resolution is then expected to be performed by the user’s application. In case the filter value is set greater than 0x05DC, the Ethernet type field is considered to represent protocol identification and only frames that are equal to the set value are to be passed to the file descriptor.

All above-set configuration options have a getter counterpart option to read the current settings.

**Warning:** The file descriptor needs to be firstly bounded to a specific Network Interface by L2TAP_S_INTF_DEVICE or L2TAP_S_DEVICE_DRV_HNDL to make L2TAP_S_RCV_FILTER option available.

**Note:** VLAN-tagged frames are currently not recognized. If the user needs to process VLAN-tagged frames, they need a set filter to be equal to the VLAN tag (i.e., 0x8100 or 0x88A8) and process the VLAN-tagged frames in the user application.

**Note:** L2TAP_S_DEVICE_DRV_HNDL is particularly useful when the user’s application does not require the usage of an IP stack and so ESP-NETIF is not required to be initialized too. As a result, Network Interface cannot be identified by its if_key and hence it needs to be identified directly by its IO Driver handle.

On success, ioctl() returns 0. On error, -1 is returned, and errno is set to indicate the error.

* EBADF - not a valid file descriptor.
* EACCES - options change is denied in this state (e.g., file descriptor has not been bounded to Network interface yet).
* EINVAL - invalid configuration argument. Ethernet type filter is already used by other file descriptors on that same Network interface.
* ENODEV - no such Network Interface which is tried to be assigned to the file descriptor exists.
* ENOSYS - unsupported operation, passed configuration option does not exist.

fcntl() fcntl() is used to manipulate with properties of opened ESP-NETIF L2 TAP file descriptor.

The following commands manipulate the status flags associated with the file descriptor:
Chapter 2. API Reference

- **F_GETFD** - the function returns the file descriptor flags, and the third argument is ignored.
- **F_SETFD** - sets the file descriptor flags to the value specified by the third argument. Zero is returned.

On success, `ioctl()` returns 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.
* ENOSYS - unsupported command.

`read()`  
Opened and configured ESP-NETIF L2 TAP file descriptor can be accessed by `read()` to get inbound frames. The read operation can be either blocking or non-blocking based on the actual state of the `O_NONBLOCK` file status flag. When the file status flag is set to blocking, the read operation waits until a frame is received and the context is switched to other tasks. When the file status flag is set to non-blocking, the read operation returns immediately. In such case, either a frame is returned if it was already queued or the function indicates the queue is empty. The number of queued frames associated with one file descriptor is limited by `CONFIG_ESP_NETIF_L2_TAP_RX_QUEUE_SIZE` Kconfig option. Once the number of queued frames reached a configured threshold, the newly arrived frames are dropped until the queue has enough room to accept incoming traffic (Tail Drop queue management).

On success, `read()` returns the number of bytes read. Zero is returned when the size of the destination buffer is 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.
* EAGAIN - the file descriptor has been marked non-blocking (`O_NONBLOCK`), and the read would block.

`write()`  
A raw Data Link Layer frame can be sent to Network Interface via opened and configured ESP-NETIF L2 TAP file descriptor. The user's application is responsible to construct the whole frame except for fields which are added automatically by the physical interface device. The following fields need to be constructed by the user's application in case of an Ethernet link: source/destination MAC addresses, Ethernet type, actual protocol header, and user data. The length of these fields is as follows:

<table>
<thead>
<tr>
<th>Destination MAC</th>
<th>Source MAC</th>
<th>Type/Length</th>
<th>Payload (protocol header/data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 B</td>
<td>6 B</td>
<td>2 B</td>
<td>0-1486 B</td>
</tr>
</tbody>
</table>

In other words, there is no additional frame processing performed by the ESP-NETIF L2 TAP interface. It only checks the Ethernet type of the frame is the same as the filter configured in the file descriptor. If the Ethernet type is different, an error is returned and the frame is not sent. Note that the `write()` may block in the current implementation when accessing a Network interface since it is a shared resource among multiple ESP-NETIF L2 TAP file descriptors and IP stack, and there is currently no queuing mechanism deployed.

On success, `write()` returns the number of bytes written. Zero is returned when the size of the input buffer is 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.
* EBADMSG - The Ethernet type of the frame is different from the file descriptor configured filter.
* EIO - Network interface not available or busy.

`close()`  
Opened ESP-NETIF L2 TAP file descriptor can be closed by the `close()` to free its allocated resources. The ESP-NETIF L2 TAP implementation of `close()` may block. On the other hand, it is thread-safe and can be called from a different task than the file descriptor is actually used. If such a situation occurs and one task is blocked in the I/O operation and another task tries to close the file descriptor, the first task is unblocked. The first task read operation then ends with an error.

On success, `close()` returns zero. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.

**select()**  Select is used in a standard way, just `CONFIG_VFS_SUPPORT_SELECT` needs to be enabled to make the `select()` function available.

**SNTP API**  You can find a brief introduction to SNTP in general, its initialization code, and basic modes in Section **SNTP Time Synchronization** in System Time.

This section provides more details about specific use cases of the SNTP service, with statically configured servers, or use the DHCP-provided servers, or both. The workflow is usually very simple:

1) Initialize and configure the service using `esp_netif_sntp_init()`. 
2) Start the service via `esp_netif_sntp_start()`. This step is not needed if we auto-started the service in the previous step (default). It is useful to start the service explicitly after connecting if we want to use the DHCP-obtained NTP servers. Please note, this option needs to be enabled before connecting, but the SNTP service should be started after.
3) Wait for the system time to synchronize using `esp_netif_sntp_sync_wait()` (only if needed).
4) Stop and destroy the service using `esp_netif_sntp_deinit()`.

**Basic Mode with Statically Defined Server(s)**  Initialize the module with the default configuration after connecting to the network. Note that it is possible to provide multiple NTP servers in the configuration struct:

```c
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE(2, ESP_SNTP_SERVER_LIST("time.windows.com", "pool.ntp.org"));
esp_netif_sntp_init(&config);
```

**Note:** If we want to configure multiple SNTP servers, we have to update the lwIP configuration `CONFIG_LWIP_SNTP_MAX_SERVERS`.

**Use DHCP-Obtained SNTP Server(s)**  First of all, we have to enable the lwIP configuration option `CONFIG_LWIP_DHCP_GET_NTP_SRV`. Then we have to initialize the SNTP module with the DHCP option and without the NTP server:

```c
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE(0, {});
config.start = false; // start the SNTP service explicitly
config.server_from_dhcp = true; // accept the NTP offer from the DHCP
esp_netif_sntp_init(&config);
```

Then, once we are connected, we could start the service using:

```c
esp_netif_sntp_start();
```

**Note:** It is also possible to start the service during initialization (default `config.start=true`). This would likely to cause the initial SNTP request to fail (since we are not connected yet) and lead to some back-off time for subsequent requests.

**Use Both Static and Dynamic Servers**  Very similar to the scenario above (DHCP provided SNTP server), but in this configuration, we need to make sure that the static server configuration is refreshed when obtaining NTP servers by DHCP. The underlying lwIP code cleans up the rest of the list of NTP servers when the DHCP-provided information gets accepted. Thus the ESP-NETIF SNTP module saves the statically configured server(s) and reconfigures them after obtaining a DHCP lease.
Chapter 2. API Reference

The typical configuration now looks as per below, providing the specific IP_EVENT to update the config and index of the first server to reconfigure (for example setting config.index_of_first_server=1 would keep the DHCP provided server at index 0, and the statically configured server at index 1).

```c
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG("pool.ntp.org");
config.start = false; // start the SNTP service explicitly_
    (after connecting)
config.server_from_dhcp = true; // accept the NTP offers from DHCP_
    server
config.renew_servers_after_new_IP = true; // let esp-netif update the configured_
    SNTP server(s) after receiving the DHCP lease
config.index_of_first_server = 1; // updates from server num 1, leaving_
    server 0 (from DHCP) intact
config.ip_event_to_renew = IP_EVENT_STA_GOT_IP; // IP event on which we refresh_
    the configuration
```

Then we start the service normally with `esp_netif_sntp_start()`.

ESP-NETIF Programmer’s Manual Please refer to the following example to understand the initialization process of the default interface:

- Wi-Fi Station: wifi/getting_started/station/main/station_example_main.c
- Ethernet: ethernet/basic/main/ethernet_example_main.c
- L2TAP: protocols/l2tap/main/l2tap_main.c
- Wi-Fi Access Point: wifi/getting_started/softAP/main/softap_example_main.c

For more specific cases, please consult this guide: ESP-NETIF Custom I/O Driver.

Wi-Fi Default Initialization The initialization code as well as registering event handlers for default interfaces, such as softAP and station, are provided in separate APIs to facilitate simple startup code for most applications:

- `esp_netif_create_default_wifi_sta()`
- `esp_netif_create_default_wifi_ap()`

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, as a consequence, which means that:

- The created object has to be destroyed if a network de-initialization is provided by an application using `esp_netif_destroy_default_wifi()`.
- These `default` interfaces must not be created multiple times unless the created handle is deleted using `esp_netif_destroy()`.
- When using Wi-Fi in AP+STA mode, both these interfaces have to be created.

API Reference

Header File

- `components/esp_netif/include/esp_netif.h`
- This header file can be included with:

  ```c
  #include "esp_netif.h"
  ```

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

  ```cmake
  REQUIRES esp_netif
  ```
  or
Chapter 2. API Reference

```c
PRIV_REQUIRES esp_netif
```

**Functions**

```c
esp_err_t esp_netif_init (void)
```

Initializes the underlying TCP/IP stack.

**Note:** This function should be called exactly once from application code, when the application starts up.

**Returns**

- ESP_OK on success
- ESP_FAIL if initializing failed

```c
esp_err_t esp_netif_deinit (void)
```

Deinitializes the esp-netif component (and the underlying TCP/IP stack)

**Note:** Deinitialization is **not** supported yet

**Returns**

- ESP_ERR_INVALID_STATE if esp_netif not initialized
- ESP_ERR_NOT_SUPPORTED otherwise

```c
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.

**Parameters**

- `esp_netif_config` - pointer to esp-netif configuration

**Returns**

- pointer to esp-netif object on success
- NULL otherwise

```c
void esp_netif_destroy (esp_netif_t *esp_netif)
```

Destroys the esp_netif object.

**Parameters**

- `esp_netif` - [in] pointer to the object to be deleted

```c
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.

**Parameters**

- `esp_netif` - [inout] pointer to the object to be configured
- `driver_config` - [in] pointer to sp_netif io driver related configuration

**Returns**

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS if invalid parameters provided

```c
esp_err_t esp_netif_attach (esp_netif_t *esp_netif, esp_netif_iodriver_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)

**Parameters**

- `esp_netif` - [inout] pointer to esp_netif object to be attached
- `driver_handle` - [in] pointer to the driver handle

**Returns**
• ESP_OK on success
• ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED if driver’s pot_attach callback failed

*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.

**Parameters**

- `esp_netif` [in] Handle to esp-netif instance
- `buffer` [in] Received data
- `len` [in] Length of the data frame
- `eb` [in] Pointer to internal buffer (used in Wi-Fi driver)

**Returns**

- ESP_OK

*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**

- `esp_netif` [in] Handle to esp-netif instance
- `base` - The base type of the event
- `event_id` - The specific ID of the event
- `data` - Optional data associated with the event

*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**

- `esp_netif` [in] Handle to esp-netif instance
- `base` - The base type of the event
- `event_id` - The specific ID of the event
- `data` - Optional data associated with the event

*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**

- `esp_netif` [in] Handle to esp-netif instance
- `base` - The base type of the event
- `event_id` - The specific ID of the event
- `data` - Optional data associated with the event
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
- **esp_netif** `[in]` Handle to esp-netif instance
- **base** The base type of the event
- **event_id** The specific ID of the event
- **data** Optional data associated with the event

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
- **esp_netif** `[in]` Handle to esp-netif instance
- **base** The base type of the event
- **event_id** The specific ID of the event
- **data** Optional data associated with the event

void esp_netif_action_join_ip6_multicast_group (void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 multicast group join.

**Note:** This API can be directly used as event handler

### Parameters
- **esp_netif** `[in]` Handle to esp-netif instance
- **base** The base type of the event
- **event_id** The specific ID of the event
- **data** Optional data associated with the event

void esp_netif_action_leave_ip6_multicast_group (void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 multicast group leave.

**Note:** This API can be directly used as event handler

### Parameters
- **esp_netif** `[in]` Handle to esp-netif instance
- **base** The base type of the event
- **event_id** The specific ID of the event
- **data** Optional data associated with the event

void esp_netif_action_add_ip6_address (void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 address added by the underlying stack.
Chapter 2. API Reference

### esp_netif_action_remove_ip6_address

```c
void esp_netif_action_remove_ip6_address (void*esp_netif, esp_event_base_t base, int32_t event_id, void*data)
```

Default building block for network interface action upon IPv6 address removed by the underlying stack.

**Note:** This API can be directly used as event handler

#### Parameters
- `esp_netif` - [in] Handle to esp-netif instance
- `base` - The base type of the event
- `event_id` - The specific ID of the event
- `data` - Optional data associated with the event

### esp_netif_set_default_netif

```c
esp_err_t esp_netif_set_default_netif (esp_netif_t*esp_netif)
```

Manual configuration of the default netif.

This API overrides the automatic configuration of the default interface based on the route_prio If the selected netif is set default using this API, no other interface could be set-default disregarding its route_prio number (unless the selected netif gets destroyed)

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance

**Returns**
- ESP_OK on success

### esp_netif_get_default_netif

```c
esp_netif_t* esp_netif_get_default_netif (void)
```

Getter function of the default netif.

This API returns the selected default netif.

**Returns**
- Handle to esp-netif instance of the default netif.

### esp_netif_join_ip6_multicast_group

```c
esp_err_t esp_netif_join_ip6_multicast_group (esp_netif_t*esp_netif, const esp_ip6_addr_t*addr)
```

Cause the TCP/IP stack to join a IPv6 multicast group.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance
- `addr` - [in] The multicast group to join

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_MLD6_FAILED
- ESP_ERR_NO_MEM

### esp_netif_leave_ip6_multicast_group

```c
esp_err_t esp_netif_leave_ip6_multicast_group (esp_netif_t*esp_netif, const esp_ip6_addr_t*addr)
```

Cause the TCP/IP stack to leave a IPv6 multicast group.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance
- `addr` - [in] The multicast group to leave

**Returns**
• ESP_OK
• ESP_ERR_ESP_NETIF_INVALID_PARAMS
• ESP_ERR_ESP_NETIF_MLD6_FAILED
• ESP_ERR_NO_MEM

```c
esp_err_t esp_netif_set_mac(esp_netif_t *esp_netif, uint8_t mac[])
```

Set the mac address for the interface instance.

**Parameters**
- `esp_netif` [in] Handle to esp-netif instance
- `mac` [in] Desired mac address for the related network interface

**Returns**
- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_NOT_SUPPORTED - mac not supported on this interface

```c
esp_err_t esp_netif_get_mac(esp_netif_t *esp_netif, uint8_t mac[])
```

Get the mac address for the interface instance.

**Parameters**
- `esp_netif` [in] Handle to esp-netif instance
- `mac` [out] Resultant mac address for the related network interface

**Returns**
- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_NOT_SUPPORTED - mac not supported on this interface

```c
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.

**Parameters**
- `esp_netif` [in] Handle to esp-netif instance

**Returns**
- ESP_OK - success
- ESP.ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP.ERR_ESP_NETIF_INVALID_PARAMS - parameter error

```c
esp_err_t esp_netif_get_hostname(esp_netif_t *esp_netif, const char **hostname)
```

Get interface hostname.

**Parameters**
- `esp_netif` [in] Handle to esp-netif instance
- `hostname` [out] 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).

**Returns**
- ESP_OK - success
- ESP.ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP.ERR_ESP_NETIF_INVALID_PARAMS - parameter error

```c
bool esp_netif_is_netif_up(esp_netif_t *esp_netif)
```

Test if supplied interface is up or down.

**Parameters**
- `esp_netif` [in] Handle to esp-netif instance

**Returns**
- true - Interface is up
- false - Interface is down
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

Parameters
• esp_netif – [in] Handle to esp-netif instance
• ip_info – [out] If successful, IP information will be returned in this argument.

Returns
• ESP_OK
• ESP_ERR_ESP_NETIF_INVALID_PARAMS

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.

Parameters
• esp_netif – [in] Handle to esp-netif instance
• ip_info – [out] If successful, IP information will be returned in this argument.

Returns
• ESP_OK
• ESP_ERR_ESP_NETIF_INVALID_PARAMS

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.

Parameters
• esp_netif – [in] Handle to esp-netif instance
• ip_info – [in] IP information to set on the specified interface

Returns
• ESP_OK
• ESP_ERR_ESP_NETIF_INVALID_PARAMS
• ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED If DHCP server or client is still running

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.

**Parameters**
- esp_netif -[in] Handle to esp-netif instance
- ip_info -[in] Store the old IP information for the specified interface

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

```c
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

**Parameters**
- esp_netif -[in] Handle to esp-netif instance

**Returns**
- implementation specific index of interface represented with supplied esp_netif

```c
esp_err_t esp_netif_get_netif_impl_name(esp_netif_t *esp_netif, char *name)
```
Get net interface name from network stack implementation.

**Note:** This name could be used in `setsockopt()` to bind socket with appropriate interface

**Parameters**
- esp_netif -[in] Handle to esp-netif instance
- name -[out] Interface name as specified in underlying TCP/IP stack. Note that the actual name will be copied to the specified buffer, which must be allocated to hold maximum interface name size (6 characters for lwIP)

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

```c
esp_err_t esp_netif_napt_enable(esp_netif_t *esp_netif)
```
Enable NAPT on an interface.

**Note:** Enable operation can be performed only on one interface at a time. NAPT cannot be enabled on multiple interfaces according to this implementation.

**Parameters**
- esp_netif -[in] Handle to esp-netif instance

**Returns**
- ESP_OK
- ESP_FAIL
- ESP_ERR_NOT_SUPPORTED

```c
esp_err_t esp_netif_napt_disable(esp_netif_t *esp_netif)
```
Disable NAPT on an interface.

**Parameters**
- esp_netif -[in] Handle to esp-netif instance

**Returns**
- ESP_OK
- ESP_FAIL
- ESP_ERR_NOT_SUPPORTED
**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.

**Parameters**
- `esp_netif` *[in]* Handle to esp-netif instance
- `opt_op` *[in]* ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- `opt_id` *[in]* Option index to get or set, must be one of the supported enum values.
- `opt_val` *[inout]* Pointer to the option parameter.
- `opt_len` *[in]* Length of the option parameter.

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

**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.

**Parameters**
- `esp_netif` *[in]* Handle to esp-netif instance
- `opt_op` *[in]* ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- `opt_id` *[in]* Option index to get or set, must be one of the supported enum values.
- `opt_val` *[inout]* Pointer to the option parameter.
- `opt_len` *[in]* Length of the option parameter.

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

**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.

**Parameters** `esp_netif` *[in]* Handle to esp-netif instance

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED
- ESP_ERR_ESP_NETIF_DHCPC_START_FAILED

**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.

**Parameters** `esp_netif` *[in]* Handle to esp-netif instance

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
Chapter 2. API Reference

- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

**esp_err_t esp_netif_dhcpc_get_status(esp_netif_t *esp_netif, esp_netif_dhcp_status_t *status)**

Get DHCP client status.

**Parameters**
- esp_netif – [in] Handle to esp-netif instance
- status – [out] If successful, the status of DHCP client will be returned in this argument.

**Returns**
- ESP_OK

**esp_err_t esp_netif_dhcps_get_status(esp_netif_t *esp_netif, esp_netif_dhcp_status_t *status)**

Get DHCP Server status.

**Parameters**
- esp_netif – [in] Handle to esp-netif instance
- status – [out] If successful, the status of the DHCP server will be returned in this argument.

**Returns**
- ESP_OK

**esp_err_t esp_netif_dhcps_start(esp_netif_t *esp_netif)**

Start DHCP server (only if enabled in interface object)

**Parameters**
- esp_netif – [in] Handle to esp-netif instance

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

**esp_err_t esp_netif_dhcps_stop(esp_netif_t *esp_netif)**

Stop DHCP server (only if enabled in interface object)

**Parameters**
- esp_netif – [in] Handle to esp-netif instance

**Returns**
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

**esp_err_t esp_netif_dhcps_get_clients_by_mac(esp_netif_t *esp_netif, int num, esp_netif_pair_mac_ip_t *mac_ip_pair)**

Populate IP addresses of clients connected to DHCP server listed by their MAC addresses.

**Parameters**
- esp_netif – [in] Handle to esp-netif instance
- num – [in] Number of clients with specified MAC addresses in the array of pairs
- mac_ip_pair – [inout] Array of pairs of MAC and IP addresses (MAC are inputs, IP outputs)

**Returns**
- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS on invalid params
- ESP_ERR_NOT_SUPPORTED if DHCP server not enabled

**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 DHCP server itself.
- This function can override it by setting server type ESP_NETIF_DNS_MAIN.
- Other DNS Server types are not supported for the DHCP server.
- To propagate the DNS info to client, please stop the DHCP server before using this API.

**Parameters**

- `esp_netif` [-in] Handle to esp-netif instance
- `type` [-in] Type of DNS Server to set: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- `dns` [-in] DNS Server address to set

**Returns**

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

```c
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.

**Parameters**

- `esp_netif` [-in] Handle to esp-netif instance
- `type` [-in] Type of DNS Server to get: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- `dns` [-out] DNS Server result is written here on success

**Returns**

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

```c
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.

**Parameters**

- `esp_netif` [-in] Handle to esp-netif instance

**Returns**

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

```c
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.

**Parameters**

- `esp_netif` [-in] Handle to esp-netif instance
- `if_ip6` [-out] IPv6 information will be returned in this argument if successful.

**Returns**

- 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.

```c
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.

**Parameters**

- `esp_netif` [in] Handle to esp-netif instance
- `if_ip6` [out] IPv6 information will be returned in this argument if successful.

**Returns**

- 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.

```c
int esp_netif_get_all_ip6(esp_netif_t *esp_netif, esp_ip6_addr_t if_ip6[])
```
Get all IPv6 addresses of the specified interface.

**Parameters**

- `esp_netif` [in] Handle to esp-netif instance
- `if_ip6` [out] Array of IPv6 addresses will be copied to the argument

**Returns**

number of returned IPv6 addresses

```c
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**

- `addr` [out] IP address to be set
- `a` – the first octet (127 for IP 127.0.0.1)
- `b` –
- `c` –
- `d` –

```c
char *esp_ip4addr_ntoa(const esp_ip4_addr_t *addr, char *buf, int buflen)
```
Converts numeric IPv4 address into decimal dotted ASCII representation.

**Parameters**

- `addr` – ip address in network order to convert
- `buf` – target buffer where the string is stored
- `buflen` – length of buf

**Returns**

either pointer to buf which now holds the ASCII representation of addr or NULL if buf was too small

```c
uint32_t esp_ip4addr_aton(const char *addr)
```
Ascii internet address interpretation routine The value returned is in network order.

**Parameters**

- `addr` – IP address in ascii representation (e.g. “127.0.0.1”)

**Returns**

ip address in network order

```c
esp_err_t esp_netif_str_to_ip4(const char *src, esp_ip4_addr_t *dst)
```
Converts Ascii internet IPv4 address into esp_ip4_addr_t.

**Parameters**

- `src` [in] IPv4 address in ascii representation (e.g. “127.0.0.1”)
- `dst` [out] Address of the target esp_ip4_addr_t structure to receive converted address

**Returns**

- ESP_OK on success
- ESP_FAIL if conversion failed
- ESP_ERR_INVALID_ARG if invalid parameter is passed into
**esp_err_t esp_netif_str_to_ip6 (const char*src, esp_ip6_addr_t *dst)**

Converts Ascii internet IPv6 address into esp_ip4_addr_t. Zeros in the IP address can be stripped or completely ommited: “2001:db8:85a3:0:0:0:2:1” or “2001:db8::2:1”

**Parameters**
- `src` - [in] IPv6 address in ascii representation (e.g. “2001:db8:85a3:0:0:0:2:1”)
- `dst` - [out] Address of the target esp_ip6_addr_t structure to receive converted address

**Returns**
- ESP_OK on success
- ESP_FAIL if conversion failed
- ESP_ERR_INVALID_ARG if invalid parameter is passed into

**esp_netif_iodevice_handle esp_netif_get_iodevice (esp_netif_t *esp_netif)**

Gets media device handle for this esp-netif instance.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance

**Returns**
- opaque pointer of related IO driver

**esp_netif_iodevice_handle esp_netif_get_iodevice_from_ifkey (const char*if_key)**

Searches over a list of created objects to find an instance with supplied if key.

**Parameters**
- `if_key` - Textual description of network interface

**Returns**
- Handle to esp-netif instance

**esp_netif_flags_t esp_netif_get_flags (esp_netif_t *esp_netif)**

Returns configured flags for this interface.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance

**Returns**
- Configuration flags

**const char* esp_netif_get_ifkey (esp_netif_t *esp_netif)**

Returns configured interface key for this esp-netif instance.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance

**Returns**
- Textual description of related interface

**const char* esp_netif_get_desc (esp_netif_t *esp_netif)**

Returns configured interface type for this esp-netif instance.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance

**Returns**
- Enumerated type of this interface, such as station, AP, ethernet

**int esp_netif_get_route_prio (esp_netif_t *esp_netif)**

Returns configured routing priority number.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance

**Returns**
- Integer representing the instance’s route-prio, or -1 if invalid parameters

**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.

**Parameters**
- `esp_netif` - [in] Handle to esp-netif instance
- `event_type` - (either get or lost IP)

**Returns**
- 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

**esp_netif_t *esp_netif_next (esp_netif_t *esp_netif)**

Iterates over list of interfaces. Returns first netif if NULL given as parameter.

**Note:** This API doesn’t lock the list, nor the TCPIP context, as this it’s usually required to get atomic access between iteration steps rather that within a single iteration. Therefore it is recommended to iterate over
Chapter 2. API Reference

the interfaces inside `esp_netif_tcpip_exec()`

---

**Note:** This API is deprecated. Please use `esp_netif_next_unsafe()` directly if all the system interfaces are under your control and you can safely iterate over them. Otherwise, iterate over interfaces using `esp_netif_tcpip_exec()`, or use `esp_netif_find_if()` to search in the list of netifs with defined predicate.

---

**Parameters** `esp_netif` -[in] Handle to esp-netif instance  
**Returns** First netif from the list if supplied parameter is NULL, next one otherwise

```c
esp_netif_t* esp_netif_next_unsafe(esp_netif_t *esp_netif)
```

Iterates over list of interfaces without list locking. Returns first netif if NULL given as parameter.

Used for bulk search loops within TCPIP context, e.g. using `esp_netif_tcpip_exec()`, or if we’re sure that the iteration is safe from our application perspective (e.g. no interface is removed between iterations)

**Parameters** `esp_netif` -[in] Handle to esp-netif instance  
**Returns** First netif from the list if supplied parameter is NULL, next one otherwise

```c
esp_netif_t* esp_netif_find_if(esp_netif_find_predicate_t fn, void *ctx)
```

Return a netif pointer for the first interface that meets criteria defined by the callback.

**Parameters**
- `fn` - Predicate function returning true for the desired interface  
- `ctx` - Context pointer passed to the predicate, typically a descriptor to compare with

**Returns** valid netif pointer if found, NULL if not

```c
size_t esp_netif_get_nr_of_ifs(void)
```

Returns number of registered esp_netif objects.

**Returns** Number of esp_netifs

```c
void esp_netif_netstack_buf_ref(void*netstack_buf)
```

Increase the reference counter of net stack buffer

**Parameters** `netstack_buf` -[in] the net stack buffer

```c
void esp_netif_netstack_buf_free(void*netstack_buf)
```

Free the netstack buffer

**Parameters** `netstack_buf` -[in] the net stack buffer

```c
esp_err_t esp_netif_tcpip_exec(esp_netif_callback_fn fn, void *ctx)
```

Utility to execute the supplied callback in TCP/IP context.

**Parameters**
- `fn` - Pointer to the callback  
- `ctx` - Parameter to the callback

**Returns** The error code (esp_err_t) returned by the callback

---

**Type Definitions**

```c
typedef bool (*)(esp_netif_find_predicate_t)(esp_netif_t *netif, void *ctx)
```

Predicate callback for `esp_netif_find_if()` used to find interface which meets defined criteria.

```c
typedef esp_err_t (*)(esp_netif_callback_fn)(void *ctx)
```

TCP/IP thread safe callback used with `esp_netif_tcpip_exec()`
Header File

- components/esp_netif/include/esp_netif_sntp.h
- This header file can be included with:

```c
#include "esp_netif_sntp.h"
```

- This header file is a part of the API provided by the esp_netif component. To declare that your component depends on esp_netif, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_netif
```

or

```cmake
PRIV_REQUIRES esp_netif
```

Functions

**esp_err_t esp_netif_sntp_init** (const esp_sntp_config_t *config)

Initialize SNTP with supplied config struct.

- **Parameters** config – Config struct
- **Returns** ESP_OK on success

**esp_err_t esp_netif_sntp_start** (void)

Start SNTP service if it wasn’t started during init (config.start = false) or restart it if already started.

- **Returns** ESP_OK on success

**void esp_netif_sntp_deinit** (void)

Deinitialize esp_netif SNTP module.

**esp_err_t esp_netif_sntp_sync_wait** (TickType_t tout)

Wait for time sync event.

- **Parameters** tout – Specified timeout in RTOS ticks
- **Returns** ESP_TIMEOUT if sync event didn’t came within the timeout ESP_ERR_NOT_FINISHED if the sync event came, but we’re in smooth update mode and still in progress (SNTP_SYNC_STATUS_IN_PROGRESS) ESP_OK if time sync’ed

Structures

**struct esp_sntp_config**

SNTP configuration struct.

Public Members

- **bool smooth_sync**
  set to true if smooth sync required

- **bool server_from_dhcp**
  set to true to request NTP server config from DHCP

- **bool wait_for_sync**
  if true, we create a semaphore to signal time sync event

- **bool start**
  set to true to automatically start the SNTP service
### esp_sntp_time_cb_t sync_cb

optionally sets callback function on time sync event

bool renew_servers_after_new_IP

this is used to refresh server list if NTP provided by DHCP (which cleans other pre-configured servers)

**ip_event_t ip_event_to_renew**

set the IP event id on which we refresh server list (if renew_servers_after_new_IP=true)

size_t index_of_first_server

refresh server list after this server (if renew_servers_after_new_IP=true)

size_t num_of_servers

number of preconfigured NTP servers

const char *servers[1]

list of servers

### Macros

**ESP_Sntp_SERVER_LIST** (...)

Utility macro for providing multiple servers in parentheses.

**ESP_NETIF_Sntp_DEFAULT_CONFIG_MULTIPLE** (servers_in_list, list_of_servers)

Default configuration to init SNTP with multiple servers.

**Parameters**

- **servers_in_list** - Number of servers in the list
- **list_of_servers** - List of servers (use ESP_Sntp_SERVER_LIST(...))

**ESP_NETIF_Sntp_DEFAULT_CONFIG** (server)

Default configuration with a single server.

### Type Definitions

typedef void (*esp_sntp_time_cb_t)(struct timeval *tv)

Time sync notification function.

typedef struct esp_sntp_config esp_sntp_config_t

SNTP configuration struct.

### Header File

- components/esp_netif/include/esp_netif_types.h

This header file can be included with:

```c
#include "esp_netif_types.h"
```

- This header file is a part of the API provided by the esp_netif component. To declare that your component depends on esp_netif, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_netif
```

or

```cmake
PRIV_REQUIRES esp_netif
```
Chapter 2. API Reference

Structures

struct esp_netif_dns_info_t
    DNS server info.

    Public Members

    esp_ip_addr_t ip
        IPV4 address of DNS server

struct esp_netif_ip_info_t
    Event structure for IP_EVENT_STA_GOT_IP, IP_EVENT_ETH_GOT_IP events

    Public Members

    esp_ip4_addr_t ip
        Interface IPV4 address

    esp_ip4_addr_t netmask
        Interface IPV4 netmask

    esp_ip4_addr_t gw
        Interface IPV4 gateway address

struct esp_netif_ip6_info_t
    IPV6 IP address information.

    Public Members

    esp_ip6_addr_t ip
        Interface IPV6 address

struct ip_event_got_ip_t
    Event structure for IP_EVENT_GOT_IP event.

    Public Members

    esp_netif_t *esp_netif
        Pointer to corresponding esp-netif object

    esp_netif_ip_info_t ip_info
        IP address, netmask, gateway IP address

    bool ip_changed
        Whether the assigned IP has changed or not

struct ip_event_got_ip6_t
    Event structure for IP_EVENT_GOT_IP6 event
Public Members

`esp_netif_t *esp_netif`
  Pointer to corresponding esp-netif object

`esp_netif_ip6_info_t ip6_info`
  IPv6 address of the interface

`int ip_index`
  IPv6 address index

`struct ip_event_add_ip6_t`
  Event structure for ADD_IP6 event

Public Members

`esp_ip6_addr_t addr`
  The address to be added to the interface

`bool preferred`
  The default preference of the address

`struct ip_event_ap_staipassigned_t`
  Event structure for IP_EVENT_AP_STAIPASSIGNED event

Public Members

`esp_netif_t *esp_netif`
  Pointer to the associated netif handle

`esp_ip4_addr_t ip`
  IP address which was assigned to the station

`uint8_t mac[6]`
  MAC address of the connected client

`struct bridgeif_config`
  LwIP bridge configuration

Public Members

`uint16_t max_fdb_dyn_entries`
  maximum number of entries in dynamic forwarding database

`uint16_t max_fdb_sta_entries`
  maximum number of entries in static forwarding database
uint8_t \textbf{max\_ports}
maximum number of ports the bridge can consist of

\begin{verbatim}
struct \textbf{esp\_netif\_inherent\_config}
ESP-netif inherent config parameters.

\textbf{Public Members}

\textit{esp\_netif\_flags\_t flags}
flags that define esp-netif behavior

uint8_t \textbf{mac[6]}
initial mac address for this interface

const \textit{esp\_netif\_ip\_info\_t \*ip\_info}
initial ip address for this interface

uint32_t \textbf{get\_ip\_event}
\begin{verbatim}
 event id to be raised when interface gets an IP
\end{verbatim}

uint32_t \textbf{lost\_ip\_event}
\begin{verbatim}
 event id to be raised when interface losts its IP
\end{verbatim}

const char \textbf{*if\_key}
\begin{verbatim}
 string identifier of the interface
\end{verbatim}

const char \textbf{*if\_desc}
\begin{verbatim}
 textual description of the interface
\end{verbatim}

int \textbf{route\_prio}
\begin{verbatim}
 numeric priority of this interface to become a default routing if (if other netifs are up). A higher value of route\_prio indicates a higher priority
\end{verbatim}

\textit{bridgeif\_config\_t \*bridge\_info}
\begin{verbatim}
 LwIP bridge configuration
\end{verbatim}

\textbf{struct \textbf{esp\_netif\_driver\_base\_s}}
ESP-netif driver base handle.

\textbf{Public Members}

\textit{esp\_err\_t (*post\_attach)(esp\_netif\_t \*netif, esp\_netif\_iodriver\_handle h)}
\begin{verbatim}
 post attach function pointer
\end{verbatim}

\textit{esp\_netif\_t \*netif}
\begin{verbatim}
 netif handle
\end{verbatim}

Su"
struct **esp_netif_driver_ifconfig**
Specific IO driver configuration.

**Public Members**

*esp_netif_iodriver_handle handle*
io-driver handle

*esp_err_t (*transmit)(void *h, void *buffer, size_t len)*
transmit function pointer

*esp_err_t (*transmit_wrap)(void *h, void *buffer, size_t len, void *netstack_buffer)*
transmit wrap function pointer

void (*driver_free_rx_buffer)(void *h, void *buffer)
free rx buffer function pointer

struct **esp_netif_config**
Generic esp_netif configuration.

**Public Members**

const *esp_netif_inherent_config_t *base*
base config

const *esp_netif_driver_ifconfig_t *driver*
driver config

const *esp_netif_netstack_config_t *stack*
stack config

struct **esp_netif_pair_mac_ip_t**
DHCP client’s addr info (pair of MAC and IP address)

**Public Members**

uint8_t mac[6]
Clients MAC address

*esp_ip4_addr_t ip*
Clients IP address

**Macros**

**ESP_ERR_ESP_NETIF_BASE**
Definition of ESP-NETIF based errors.
ESP_ERR_ESP_NETIF_INVALID_PARAMS
ESP_ERR_ESP_NETIF_IF_NOT_READY
ESP_ERR_ESP_NETIF_DHCPC_START_FAILED
ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED
ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
ESP_ERR_ESP_NETIF_NO_MEM
ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED
ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED
ESP_ERR_ESP_NETIF_INIT_FAILED
ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED
ESP_ERR_ESP_NETIF_MLD6_FAILED
ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED
ESP_ERR_ESP_NETIF_DHCPS_START_FAILED

ESP_NETIF_BR_FLOOD
   Definition of ESP-NETIF bridge controll.

ESP_NETIF_BR_DROP

ESP_NETIF_BR_FDW_CPU

**Type Definitions**

typedef struct esp_netif_obj **esp_netif_t**

typedef enum **esp_netif_flags** **esp_netif_flags_t**

typedef enum **esp_netif_ip_event_type** **esp_netif_ip_event_type_t**

typedef struct **bridgeif_config** **bridgeif_config_t**
   LwIP bridge configuration

.typedef struct **esp_netif_inherent_config** **esp_netif_inherent_config_t**
   ESP-netif inherent config parameters.
typedef struct esp_netif_config esp_netif_config_t

typedef void *esp_netif_iodriver_handle
   IO driver handle type.

typedef struct esp_netif_driver_base_s esp_netif_driver_base_t
   ESP-netif driver base handle.

typedef struct esp_netif_driver_ifconfig esp_netif_driver_ifconfig_t

typedef struct esp_netif_netstack_config esp_netif_netstack_config_t
   Specific L3 network stack configuration.

typedef esp_err_t (*esp_netif_receive_t)(esp_netif_t *esp_netif, void *buffer, size_t len, void *eb)
   ESP-NETIF Receive function type.

**Enumerations**

enum esp_netif_dns_type_t
   Type of DNS server.
   Values:

   enumerator ESP_NETIF_DNS_MAIN
      DNS main server address

   enumerator ESP_NETIF_DNS_BACKUP
      DNS backup server address (Wi-Fi STA and Ethernet only)

   enumerator ESP_NETIF_DNS_FALLBACK
      DNS fallback server address (Wi-Fi STA and Ethernet only)

   enumerator ESP_NETIF_DNS_MAX

enum esp_netif_dhcp_status_t
   Status of DHCP client or DHCP server.
   Values:

   enumerator ESP_NETIF_DHCP_INIT
      DHCP client/server is in initial state (not yet started)

   enumerator ESP_NETIF_DHCP_STARTED
      DHCP client/server has been started

   enumerator ESP_NETIF_DHCP_STOPPED
      DHCP client/server has been stopped

   enumerator ESP_NETIF_DHCP_STATUS_MAX
enum esp_netif_dhcp_option_mode_t
    Mode for DHCP client or DHCP server option functions.
    
    Values:
    
    enumerator ESP_NETIF_OP_START
    enumerator ESP_NETIF_OP_SET
        Set option
    enumerator ESP_NETIF_OP_GET
        Get option
    enumerator ESP_NETIF_OP_MAX

enum esp_netif_dhcp_option_id_t
    Supported options for DHCP client or DHCP server.
    
    Values:
    
    enumerator ESP_NETIF_SUBNET_MASK
        Network mask
    enumerator ESP_NETIF_DOMAIN_NAME_SERVER
        Domain name server
    enumerator ESP_NETIF_ROUTER_SOLICITATION_ADDRESS
        Solicitation router address
    enumerator ESP_NETIF_REQUESTED_IP_ADDRESS
        Request specific IP address
    enumerator ESP_NETIF_IP_ADDRESS_LEASE_TIME
        Request IP address lease time
    enumerator ESP_NETIF_IP_REQUEST_RETRY_TIME
        Request IP address retry counter
    enumerator ESP_NETIF_VENDOR_CLASS_IDENTIFIER
        Vendor Class Identifier of a DHCP client
    enumerator ESP_NETIF_VENDOR_SPECIFIC_INFO
        Vendor Specific Information of a DHCP server

enum ip_event_t
    IP event declarations
    
    Values:
    
    enumerator IP_EVENT_STA_GOT_IP
        station got IP from connected AP
enumerator **IP_EVENT_STA_LOST_IP**
station lost IP and the IP is reset to 0

enumerator **IP_EVENT_AP_STAIPASSIGNED**
soft-AP assign an IP to a connected station

enumerator **IP_EVENT_GOT_IP6**
station or ap or ethernet interface v6IP addr is preferred

enumerator **IP_EVENT_ETH_GOT_IP**
eternet got IP from connected AP

enumerator **IP_EVENT_ETH_LOST_IP**
eternet lost IP and the IP is reset to 0

enumerator **IP_EVENT_PPP_GOT_IP**
PPP interface got IP

enumerator **IP_EVENT_PPP_LOST_IP**
PPP interface lost IP

enum **esp_netif_flags**
Values:

enumerator **ESP_NETIF_DHCP_CLIENT**

enumerator **ESP_NETIF_DHCP_SERVER**

enumerator **ESP_NETIF_FLAG_AUTOUP**

enumerator **ESP_NETIF_FLAG_GARP**

enumerator **ESP_NETIF_FLAG_EVENT_IP_MODIFIED**

enumerator **ESP_NETIF_FLAG_IS_PPP**

enumerator **ESP_NETIF_FLAG_IS_BRIDGE**

enumerator **ESP_NETIF_FLAG_MLDV6_REPORT**

enum **esp_netif_ip_event_type**
Values:

enumerator **ESP_NETIF_IP_EVENT_GOT_IP**

enumerator **ESP_NETIF_IP_EVENT_LOST_IP**
Chapter 2. API Reference

Header File

- components/esp_netif/include/esp_netif_ip_addr.h
- This header file can be included with:
  ```
  #include "esp_netif_ip_addr.h"
  ```
- This header file is a part of the API provided by the esp_netif component. To declare that your component depends on esp_netif, add the following to your CMakeLists.txt:
  ```
  REQUIRES esp_netif
  ```
  or
  ```
  PRIV_REQUIRES esp_netif
  ```

Functions

```
esp_ip6_addr_type_t esp_netif_ip6_get_addr_type(esp_ip6_addr_t *ip6_addr)
```
Get the IPv6 address type.

- **Parameters**
  - `ip6_addr` [in] IPv6 type
- **Returns**
  IPv6 type in form of enum esp_ip6_addr_type_t

```
static inline void esp_netif_ip_addr_copy(esp_ip_addr_t *dest, const esp_ip_addr_t *src)
```
Copy IP addresses.

- **Parameters**
  - `dest` [out] destination IP
  - `src` [in] source IP

Structures

```
struct esp_ip6_addr
```
IPv6 address.

- **Public Members**
  ```
  uint32_t addr[4]
  ```
  IPv6 address
  ```
  uint8_t zone
  ```
  zone ID

```
struct esp_ip4_addr
```
IPv4 address.

- **Public Members**
  ```
  uint32_t addr
  ```
  IPv4 address
  ```
  struct _ip_addr
  ```
  IP address.
Chapter 2. API Reference

Public Members

\textit{esp_ip6_addr_t} \texttt{ip6}

IPv6 address type

\textit{esp_ip4_addr_t} \texttt{ip4}

IPv4 address type

union \texttt{\_ip_addr::[anonymous]} \texttt{u\_addr}

IP address union

\texttt{uint8\_t type}

ipaddress type

Macros

\texttt{esp\_netif\_htonl (x)}

\texttt{esp\_netif\_ip4\_makeu32 (a, b, c, d)}

\texttt{ESP\_IP6\_ADDR\_BLOCK1 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK2 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK3 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK4 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK5 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK6 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK7 (ip6addr)}

\texttt{ESP\_IP6\_ADDR\_BLOCK8 (ip6addr)}

\texttt{IPSTR}

\texttt{esp\_ip4\_addr\_get\_byte (ipaddr, idx)}

\texttt{esp\_ip4\_addr1 (ipaddr)}

\texttt{esp\_ip4\_addr2 (ipaddr)}

\texttt{esp\_ip4\_addr3 (ipaddr)}

\texttt{esp\_ip4\_addr4 (ipaddr)}

\texttt{esp\_ip4\_addr1\_16 (ipaddr)}

\texttt{esp\_ip4\_addr2\_16 (ipaddr)}

\texttt{esp\_ip4\_addr3\_16 (ipaddr)}

\texttt{esp\_ip4\_addr4\_16 (ipaddr)}

\texttt{IP2STR (ipaddr)}

\texttt{IPV6STR}
IPV62STR (ipaddr)

ESP_IPADDR_TYPE_V4

ESP_IPADDR_TYPE_V6

ESP_IPADDR_TYPE_ANY

ESP_IP4TOINT32 (a, b, c, d)

ESP_IP4TOADDR (a, b, c, d)

ESP_IP4ADDR_INIT (a, b, c, d)

ESP_IP6ADDR_INIT (a, b, c, d)

IP4ADDR_STRLEN_MAX

ESP_IP_IS_ANY (addr)

**Type Definitions**

typedef struct esp_ip4_addr esp_ip4_addr_t

typedef struct esp_ip6_addr esp_ip6_addr_t

typedef struct _ip_addr esp_ip_addr_t

IP address.

**Enumerations**

enum esp_ip6_addr_type_t

Values:

enumerator ESP_IP6_ADDR_IS_UNKNOWN

enumerator ESP_IP6_ADDR_IS_GLOBAL

enumerator ESP_IP6_ADDR_IS_LINK_LOCAL

enumerator ESP_IP6_ADDR_IS_SITE_LOCAL

enumerator ESP_IP6_ADDR_IS_UNIQUE_LOCAL

enumerator ESP_IP6_ADDR_IS_IPV4_MAPPED_IPV6

**Header File**

- components/esp_netif/include/esp_vfs_l2tap.h
- This header file can be included with:
#include "esp_vfs_l2tap.h"

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_netif
```

or

```cmake
PRIV_REQUIRES esp_netif
```

## Functions

### `esp_err_t esp_vfs_l2tap_intf_register (l2tap_vfs_config_t *config)`

Add L2 TAP virtual filesystem driver.

This function must be called prior usage of ESP-NETIF L2 TAP Interface

- **Parameters** `config` - L2 TAP virtual filesystem driver configuration. Default base path `/dev/net/tapis` is used when this parameter is NULL.
- **Returns** `esp_err_t`
  - ESP_OK on success

### `esp_err_t esp_vfs_l2tap_intf_unregister (const char* base_path)`

Removes L2 TAP virtual filesystem driver.

- **Parameters** `base_path` - Base path to the L2 TAP virtual filesystem driver. Default path `/dev/net/tapis` is used when this parameter is NULL.
- **Returns** `esp_err_t`
  - ESP_OK on success

### `esp_err_t esp_vfs_l2tap_eth_filter (l2tap_iodriver_handle driver_handle, void *buff, size_t *size)`

Filters received Ethernet L2 frames into L2 TAP infrastructure.

- **Parameters**
  - `driver_handle` - handle of driver at which the frame was received
  - `buff` - received L2 frame
  - `size` - input length of the L2 frame which is set to 0 when frame is filtered into L2 TAP
- **Returns** `esp_err_t`
  - ESP_OK is always returned

## Structures

```c
struct l2tap_vfs_config_t
```

L2Tap VFS config parameters.

## Public Members

```c
const char *base_path
```

vfs base path

## Macros

```c
L2TAP_VFS_DEFAULT_PATH
L2TAP_VFS_CONFIG_DEFAULT()
```
Type Definitions

typedef void *l2tap_iodriver_handle

Enumerations

enum l2tap_ioctl_opt_t

Values:

enumerator L2TAP_S_RCV_FILTER
enumerator L2TAP_G_RCV_FILTER
enumerator L2TAP_S_INTF_DEVICE
enumerator L2TAP_G_INTF_DEVICE
enumerator L2TAP_S_DEVICE_DRV_HNDL
enumerator L2TAP_G_DEVICE_DRV_HNDL

Wi-Fi Default API Reference

Header File

- components/esp_wifi/include/esp_wifi_default.h
- This header file can be included with:

```c
#include "esp_wifi_default.h"
```
- This header file is a part of the API provided by the esp_wifi component. To declare that your component depends on esp_wifi, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_wifi
```

or

```cmake
PRIV_REQUIRES esp_wifi
```

Functions

- `esp_err_t esp_netif_attach_wifi_station(esp_netif_t *esp_netif)`
  Attaches wifi station interface to supplied netif.
  
  Parameters `esp_netif` - instance to attach the wifi station to
  
  Returns
  - ESP_OK on success
  - ESP_FAIL if attach failed

- `esp_err_t esp_netif_attach_wifi_ap(esp_netif_t *esp_netif)`
  Attaches wifi soft AP interface to supplied netif.
  
  Parameters `esp_netif` - instance to attach the wifi AP to
  
  Returns
  - ESP_OK on success
  - ESP_FAIL if attach failed
Chapter 2. API Reference

**esp_err_t** `esp_wifi_set_default_wifi_sta_handlers (void)`
Sets default wifi event handlers for STA interface.

**Returns**
- 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 AP interface.

**Returns**
- ESP_OK on success, error returned from esp_event_handler_register if failed

**esp_err_t** `esp_wifi_set_default_wifi_nan_handlers (void)`
Sets default wifi event handlers for NAN interface.

**Returns**
- 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.

**Parameters**
- `esp_netif` — instance of corresponding if object

**Returns**
- ESP_OK on success, error returned from esp_event_handler_register if failed

**esp_netif_t** `*esp_netif_create_default_wifi_ap (void)`
Creates default WIFI AP. In case of any init error this API aborts.

**Note:** The API creates esp_netif object with default WiFi access point config, attaches the netif to wifi and registers wifi handlers to the default event loop. This API uses assert() to check for potential errors, so it could abort the program. (Note that the default event loop needs to be created prior to calling this API)

**Returns**
- 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.

**Note:** The API creates esp_netif object with default WiFi station config, attaches the netif to wifi and registers wifi handlers to the default event loop. This API uses assert() to check for potential errors, so it could abort the program. (Note that the default event loop needs to be created prior to calling this API)

**Returns**
- pointer to esp-netif instance

**esp_netif_t** `*esp_netif_create_default_wifi_nan (void)`
Creates default WIFI NAN. In case of any init error this API aborts.

**Note:** The API creates esp_netif object with default WiFi station config, attaches the netif to wifi and registers wifi handlers to the default event loop. (Note that the default event loop needs to be created prior to calling this API)

**Returns**
- pointer to esp-netif instance
Chapter 2. API Reference

void esp_netif_destroy_default_wifi (void *esp_netif)

Destroys default WIFI netif created with esp_netif_create_default_wifi() API.

Note: This API unregisters wifi handlers and detaches the created object from the wifi. (this function is a no-operation if esp_netif is NULL)

Parameters

esp_netif *esp_netif_destroy_default_wifi

esp_netif_t *esp_netif_create_wifi (wifi_interface_t wifi_if, const esp_netif_inherent_config_t *esp_netif_config)

Creates esp_netif WiFi object based on the custom configuration.

Attention This API DOES NOT register default handlers!

Parameters

• wifi_if [in] type of wifi interface
• esp_netif_config -inherent esp_netif configuration pointer

Returns pointer to esp-netif instance

esp_err_t esp_netif_create_default_wifi_mesh_netifs (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.

Parameters

• p_netif_sta [out] pointer where the resultant STA interface is saved (if non NULL)
• p_netif_ap [out] pointer where the resultant AP interface is saved (if non NULL)

Returns ESP_OK on success

2.4.5 IP Network Layer

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 According to the diagram shown in the ESP-NETIF Architecture part, 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:

```c
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 result, the I/O driver has to create an instance of the struct per below:

```c
typedef struct my_netif_driver_s {
    esp_netif_driver_base_t base;  /*! base structure reserved as...
    struct esp_netif_driver */
    driver_impl_t *h;  /*! handle of driver...*/
    implementation_t */
} 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:

```c
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 behavior 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:

```c
esp_err_t my_driver_netif_set_default_handlers(esp_netif_t *esp_netif, 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.

Note that ESP-IDF provides several network stack configurations for the most common network interfaces, such as for the Wi-Fi station or Ethernet. These configurations are defined in `esp_netif/include/esp_netif_defaults.h` and should be sufficient for most network drivers. In rare cases, expert users might want to define custom lwIP based interface layers; it is possible, but an explicit dependency to lwIP needs to be set.

The following API reference outlines these network stack interaction with the ESP-NETIF:

Header File

- `components/esp_netif/include/esp_netif_net_stack.h`
  - This header file can be included with:
    ```c
    #include "esp_netif_net_stack.h"
    ```
  - This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your `CMakeLists.txt`:
    ```cmake
    REQUIRE esp_netif
    ```
    or
    ```cmake
    PRIV_REQUIRES esp_netif
    ```

Functions

- `esp_netif_t *esp_netif_get_handle_from_netif_impl(void *dev)`
  - Returns esp-netif handle.
  - **Parameters**
    - `dev` — [in] opaque ptr to network interface of specific TCP/IP stack
  - **Returns** handle to related esp-netif instance

- `void *esp_netif_get_netif_impl(esp_netif_t *esp_netif)`
  - Returns network stack specific implementation handle (if supported)
  - **Parameters**
    - `esp_netif` — [in] Handle to esp-netif instance
  - **Returns** handle to related network stack netif handle

- `esp_err_t esp_netif_set_link_speed(esp_netif_t *esp_netif, uint32_t speed)`
  - Set link-speed for the specified network interface.
  - **Parameters**
    - `esp_netif` — [in] Handle to esp-netif instance
    - `speed` — [in] Link speed in bit/s
  - **Returns** ESP_OK on success

- `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.
  - **Parameters**
    - `esp_netif` — [in] Handle to esp-netif instance
    - `data` — [in] Data to be transmitted
    - `len` — [in] Length of the data frame
  - **Returns** ESP_OK on success, an error passed from the I/O driver otherwise

- `esp_err_t esp_netif_transmit_wrap(esp_netif_t *esp_netif, void *data, size_t len, void *netstack_buf)`
  - 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.
Chapter 2. API Reference

Parameters

- **esp_netif** - [in] Handle to esp-netif instance
- **data** - [in] Data to be transmitted
- **len** - [in] Length of the data frame
- **netstack_buf** - [in] net stack buffer

Returns

ESP_OK on success, an error passed from the I/O driver otherwise

```c
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 I/O driver context, i.e. to deallocate a buffer owned by io driver (when data packets were passed to higher levels to avoid copying)

Parameters

- **esp_netif** - [in] Handle to esp-netif instance
- **buffer** - [in] Rx buffer pointer

Code examples for TCP/IP socket APIs are provided in the protocols/sockets directory of ESP-IDF examples.

2.4.6 Application Layer

Documentation for Application layer network protocols (above the IP Network layer) are provided in Application Protocols.

2.5 Peripherals API

2.5.1 Analog to Digital Converter (ADC) Oneshot Mode Driver

**Introduction**

The Analog to Digital Converter is integrated on the chip and is capable of measuring analog signals from specific analog IO pins.

ESP32-S2 has two ADC unit(s), which can be used in scenario(s) like:

- Generate one-shot ADC conversion result
- Generate continuous ADC conversion results

This guide introduces ADC oneshot mode conversion.

**Functional Overview**

The following sections of this document cover the typical steps to install and operate an ADC:

- **Resource Allocation** - covers which parameters should be set up to get an ADC handle and how to recycle the resources when ADC finishes working.
- **Unit Configuration** - covers the parameters that should be set up to configure the ADC unit, so as to get ADC conversion raw result.
- **Read Conversion Result** - covers how to get ADC conversion raw result.
- **Hardware Limitations** - describes the ADC-related hardware limitations.
- **Power Management** - covers power management-related information.
- **IRAM Safe** - describes tips on how to read ADC conversion raw results when the cache is disabled.
• **Thread Safety** - lists which APIs are guaranteed to be thread-safe by the driver.

• **Kconfig Options** - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

**Resource Allocation**  
The ADC oneshot mode driver is implemented based on ESP32-S2 SAR ADC module. Different ESP chips might have different numbers of independent ADCs. From the oneshot mode driver’s point of view, an ADC instance is represented by `adc_oneshot_unit_handle_t`.

To install an ADC instance, set up the required initial configuration structure `adc_oneshot_unit_init_cfg_t`:

- `adc_oneshot_unit_init_cfg_t::unit_id` selects the ADC. Please refer to the datasheet to know dedicated analog IOs for this ADC.
- `adc_oneshot_unit_init_cfg_t::clk_src` selects the source clock of the ADC. If set to 0, the driver will fall back to using a default clock source, see `adc_oneshot_clk_src_t` to know the details.
- `adc_oneshot_unit_init_cfg_t::ulp_mode` sets if the ADC will be working under ULP mode.

After setting up the initial configurations for the ADC, call `adc_oneshot_new_unit()` with the prepared `adc_oneshot_unit_init_cfg_t`. This function will return an ADC unit handle if the allocation is successful.

This function may fail due to various errors such as invalid arguments, insufficient memory, etc. Specifically, when the to-be-allocated ADC instance is registered already, this function will return `ESP_ERR_NOT_FOUND` error. Number of available ADC(s) is recorded by `SOC_ADC_PERIPH_NUM`.

If a previously created ADC instance is no longer required, you should recycle the ADC instance by calling `adc_oneshot_del_unit()`, related hardware and software resources will be recycled as well.

**Create an ADC Unit Handle Under Normal Oneshot Mode**

```c
adc_oneshot_unit_handle_t adc1_handle;
adc_oneshot_unit_init_cfg_t init_config1 = {
    .unit_id = ADC_UNIT_1,
    .ulp_mode = ADC_ULP_MODE_DISABLE,
};
ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));
```

**Recycle the ADC Unit**

```c
ESP_ERROR_CHECK(adc_oneshot_del_unit(adc1_handle));
```

**Unit Configuration**  
After an ADC instance is created, set up the `adc_oneshot_chan_cfg_t` to configure ADC IOs to measure analog signal:

- `adc_oneshot_chan_cfg_t::atten`, ADC attenuation. Refer to TRM > On-Chip Sensor and Analog Signal Processing.
- `adc_oneshot_chan_cfg_t::bitwidth`, the bitwidth of the raw conversion result.

**Note:** For the IO corresponding ADC channel number, check datasheet to know the ADC IOs.

Additionally, `adc_continuous_io_to_channel()` and `adc_continuous_channel_to_io()` can be used to know the ADC channels and ADC IOs.

To make these settings take effect, call `adc_oneshot_config_channel()` with the above configuration structure. You should specify an ADC channel to be configured as well. Function `adc_oneshot_config_channel()` can be called multiple times to configure different ADC channels. The Driver will save each of these channel configurations internally.
Configure Two ADC Channels

```c
adc_oneshot_chan_cfg_t config = {
  .bitwidth = ADC_BITWIDTH_DEFAULT,
  .atten = ADC_ATTEN_DB_12,
};
ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, EXAMPLE_ADC1_CHAN0, &config));
ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, EXAMPLE_ADC1_CHAN1, &config));
```

**Read Conversion Result**  
After above configurations, the ADC is ready to measure the analog signal(s) from the configured ADC channel(s). Call `adc_oneshot_read()` to get the conversion raw result of an ADC channel.

- `adc_oneshot_read()` is safe to use. ADC(s) are shared by some other drivers/peripherals, see *Hardware Limitations*. This function uses mutexes to avoid concurrent hardware usage. Therefore, this function should not be used in an ISR context. This function may fail when the ADC is in use by other drivers/peripherals, and return `ESP_ERR_TIMEOUT`. Under this condition, the ADC raw result is invalid.

This function will fail due to invalid arguments.

The ADC conversion results read from this function are raw data. To calculate the voltage based on the ADC raw results, this formula can be used:

\[
V_{\text{out}} = D_{\text{out}} \times \frac{V_{\text{max}}}{D_{\text{max}}}
\]

where:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_{\text{out}}</td>
<td>Digital output result, standing for the voltage.</td>
</tr>
<tr>
<td>D_{\text{out}}</td>
<td>ADC raw digital reading result.</td>
</tr>
<tr>
<td>V_{\text{max}}</td>
<td>Maximum measurable input analog voltage, this is related to the ADC attenuation, please refer to TRM &gt; On-Chip Sensor and Analog Signal Processing.</td>
</tr>
<tr>
<td>D_{\text{max}}</td>
<td>Maximum of the output ADC raw digital reading result, which is (2^\text{bitwidth}), where \text{bitwidth} is the :cpp:member::adc_oneshot_chan_cfg_t:bitwidth configured before.</td>
</tr>
</tbody>
</table>

To do further calibration to convert the ADC raw result to voltage in mV, please refer to calibration doc *Analog to Digital Converter (ADC) Calibration Driver*.

**Read Raw Result**

```c
ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, EXAMPLE_ADC1_CHAN0, &adc_raw[0][0]));
ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1, EXAMPLE_ADC1_CHAN0, adc_raw[0][0]);
ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, EXAMPLE_ADC1_CHAN1, &adc_raw[0][1]));
ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1, EXAMPLE_ADC1_CHAN1, adc_raw[0][1]);
```

**Hardware Limitations**

- Random Number Generator (RNG) uses ADC as an input source. When ADC `adc_oneshot_read()` works, the random number generated from RNG will be less random.

- A specific ADC unit can only work under one operating mode at any one time, either continuous mode or oneshot mode. `adc_oneshot_read()` has provided the protection.

- ADC2 is also used by Wi-Fi. `adc_oneshot_read()` has provided protection between the Wi-Fi driver and ADC oneshot mode driver.
Power Management  When power management is enabled, i.e., `CONFIG_PM_ENABLE` is on, the system clock frequency may be adjusted when the system is in an idle state. However, the ADC oneshot mode driver works in a polling routine, the `adc_oneshot_read()` will poll the CPU until the function returns. During this period of time, the task in which ADC oneshot mode driver resides will not be blocked. Therefore the clock frequency is stable when reading.

IRAM Safe  By default, all the ADC oneshot mode driver APIs are not supposed to be run when the Cache is disabled. Cache may be disabled due to many reasons, such as Flash writing/erasing, OTA, etc. If these APIs execute when the Cache is disabled, you will probably see errors like `Illegal Instruction` or `Load/Store Prohibited`.

Thread Safety

- `adc_oneshot_new_unit()`
- `adc_oneshot_config_channel()`
- `adc_oneshot_read()`

Above functions are guaranteed to be thread-safe. Therefore, you can call them from different RTOS tasks without protection by extra locks.

- `adc_oneshot_del_unit()` is not thread-safe. Besides, concurrently calling this function may result in failures of the above thread-safe APIs.

Kconfig Options

- `CONFIG_ADC_ONESHOT_CTRL_FUNC_IN_IRAM` controls where to place the ADC fast read function (IRAM or Flash), see `IRAM Safe` for more details.

Application Examples

- ADC oneshot mode example: `peripherals/adc/oneshot_read`.

API Reference

Header File

- `components/hal/include/hal/adc_types.h`
- This header file can be included with:

```
#include "hal/adc_types.h"
```

Structures

struct `adc_digi_pattern_config_t`

ADC digital controller pattern configuration.

Public Members

- `uint8_t atten`
  
  Attenuation of this ADC channel.

- `uint8_t channel`
  
  ADC channel.
Chapter 2. API Reference

`uint8_t unit`
ADC unit.

`uint8_t bit_width`
ADC output bit width.

`struct adc_digi_output_data_t`
ADC digital controller (DMA mode) output data format. Used to analyze the acquired ADC (DMA) data.

**Note:** ESP32: Only `type1` is valid. ADC2 does not support DMA mode.

**Note:** ESP32-S2: Member `channel` can be used to judge the validity of the ADC data, because the role of the arbiter may get invalid ADC data.

**Public Members**

`uint16_t data`
ADC real output data info. Resolution: 12 bit.
ADC real output data info. Resolution: 11 bit.

`uint16_t channel`
ADC channel index info.
ADC channel index info. For ESP32-S2: If (channel < ADC_CHANNEL_MAX), The data is valid. If (channel > ADC_CHANNEL_MAX), The data is invalid.

`struct adc_digi_output_data_t::[anonymous]::[anonymous] type1`
ADC type1

`uint16_t unit`
ADC unit index info. 0: ADC1; 1: ADC2.

`struct adc_digi_output_data_t::[anonymous]::[anonymous] type2`
When the configured output format is 11bit.

`uint16_t val`
Raw data value

`struct adc_digi_clk_t`
ADC digital controller (DMA mode) clock system setting. Calculation formula: `controller_clk = (APLL or APB) / (div_num + div_a / div_b + 1)`.

**Note:** The clocks of the DAC digital controller use the ADC digital controller clock divider.
Public Members

```c
bool use_apll
    true: use PLL clock; false: use APB clock.
```

```c
uint32_t div_num
    Division factor. Range: 0 ~ 255. Note: When a higher frequency clock is used (the division factor is less than 9), the ADC reading value will be slightly offset.
```

```c
uint32_t div_b
    Division factor. Range: 1 ~ 63.
```

```c
uint32_t div_a
    Division factor. Range: 0 ~ 63.
```

Type Definitions

typedef `soc_periph_adc_digi_clk_src_t` `adc_oneshot_clk_src_t`
Clock source type of oneshot mode which uses digital controller.

typedef `soc_periph_adc_digi_clk_src_t` `adc_continuous_clk_src_t`
Clock source type of continuous mode which uses digital controller.

Enumerations

enum `adc_unit_t`
ADC unit.
Values:

- enumerator `ADC_UNIT_1`
  SAR ADC 1.

- enumerator `ADC_UNIT_2`
  SAR ADC 2.

enum `adc_channel_t`
ADC channels.
Values:

- enumerator `ADC_CHANNEL_0`
  ADC channel.

- enumerator `ADC_CHANNEL_1`
  ADC channel.

- enumerator `ADC_CHANNEL_2`
  ADC channel.

- enumerator `ADC_CHANNEL_3`
  ADC channel.
enumerator ADC_CHANNEL_4
ADC channel.

enumerator ADC_CHANNEL_5
ADC channel.

enumerator ADC_CHANNEL_6
ADC channel.

enumerator ADC_CHANNEL_7
ADC channel.

enumerator ADC_CHANNEL_8
ADC channel.

enumerator ADC_CHANNEL_9
ADC channel.

enum adc_atten_t
ADC attenuation parameter. Different parameters determine the range of the ADC.
Values:

enumerator ADC_ATTEN_DB_0
No input attenuation, ADC can measure up to approx.

enumerator ADC_ATTEN_DB_2_5
The input voltage of ADC will be attenuated extending the range of measurement by about 2.5 dB.

enumerator ADC_ATTEN_DB_6
The input voltage of ADC will be attenuated extending the range of measurement by about 6 dB.

enumerator ADC_ATTEN_DB_12
The input voltage of ADC will be attenuated extending the range of measurement by about 12 dB.

enumerator ADC_ATTEN_DB_11
This is deprecated, it behaves the same as ADC_ATTEN_DB_12

enum adc_bitwidth_t
Values:

enumerator ADC_BITWIDTH_DEFAULT
Default ADC output bits, max supported width will be selected.

enumerator ADC_BITWIDTH_9
ADC output width is 9Bit.

enumerator ADC_BITWIDTH_10
ADC output width is 10Bit.
enumerator **ADC_BITWIDTH_11**
ADC output width is 11Bit.

enumerator **ADC_BITWIDTH_12**
ADC output width is 12Bit.

enumerator **ADC_BITWIDTH_13**
ADC output width is 13Bit.

enum **adc_ulp_mode_t**
*Values:*

enumerator **ADC_ULP_MODE_DISABLE**
ADC ULP mode is disabled.

enumerator **ADC_ULP_MODE_FSM**
ADC is controlled by ULP FSM.

enumerator **ADC_ULP_MODE_RISCV**
ADC is controlled by ULP RISCV.

enum **adc_digi_convert_mode_t**
ADC digital controller (DMA mode) work mode.
*Values:*

enumerator **ADC_CONV_SINGLE_UNIT_1**
Only use ADC1 for conversion.

enumerator **ADC_CONV_SINGLE_UNIT_2**
Only use ADC2 for conversion.

enumerator **ADC_CONV_BOTH_UNIT**
Use Both ADC1 and ADC2 for conversion simultaneously.

enumerator **ADC_CONV_ALTER_UNIT**
Use both ADC1 and ADC2 for conversion by turn. e.g. ADC1 -> ADC2 -> ADC1 -> ADC2 ···

enum **adc_digi_output_format_t**
ADC digital controller (DMA mode) output data format option.
*Values:*

enumerator **ADC_DIGI_OUTPUT_FORMAT_TYPE1**
See **adc_digi_output_data_t.type1**

enumerator **ADC_DIGI_OUTPUT_FORMAT_TYPE2**
See **adc_digi_output_data_t.type2**
enum adc_digi_iir_filter_t
ADC IIR Filter ID.

    Values:

    enumerator ADC_DIGI_IIR_FILTER_0
        Filter 0.

    enumerator ADC_DIGI_IIR_FILTER_1
        Filter 1.

enum adc_digi_iir_filter_coeff_t
IIR Filter Coefficient.

    Values:

    enumerator ADC_DIGI_IIR_FILTER_COEFF_2
        The filter coefficient is 2.

    enumerator ADC_DIGI_IIR_FILTER_COEFF_4
        The filter coefficient is 4.

    enumerator ADC_DIGI_IIR_FILTER_COEFF_8
        The filter coefficient is 8.

    enumerator ADC_DIGI_IIR_FILTER_COEFF_16
        The filter coefficient is 16.

    enumerator ADC_DIGI_IIR_FILTER_COEFF_64
        The filter coefficient is 64.

enum adc_monitor_id_t
ADC monitor (continuous mode) ID.

    Values:

    enumerator ADC_MONITOR_0
        The monitor index 0.

    enumerator ADC_MONITOR_1
        The monitor index 1.

enum adc_monitor_mode_t
Monitor config/event mode type.

    Values:

    enumerator ADC_MONITOR_MODE_HIGH
        ADC raw_result > threshold value, monitor interrupt will be generated.

    enumerator ADC_MONITOR_MODE_LOW
        ADC raw_result < threshold value, monitor interrupt will be generated.
Header File

- components/esp_adc/include/esp_adc/adc_oneshot.h
- This header file can be included with:

```c
#include "esp_adc/adc_oneshot.h"
```

- This header file is a part of the API provided by the esp_adc component. To declare that your component depends on esp_adc, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_adc
```

or

```cmake
PRIV_REQUIRES esp_adc
```

Functions

`esp_err_t adc_oneshot_new_unit` (const `adc_oneshot_unit_init_cfg_t` *init_config, `adc_oneshot_unit_handle_t` *ret_unit)

Create a handle to a specific ADC unit.

**Note:** This API is thread-safe. For more details, see ADC programming guide

**Parameters**
- `init_config` - [in] Driver initial configurations
- `ret_unit` - [out] ADC unit handle

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_NO_MEM: No memory
- ESP_ERR_NOT_FOUND: The ADC peripheral to be claimed is already in use
- ESP_FAIL: Clock source isn’t initialised correctly

`esp_err_t adc_oneshot_config_channel` (`adc_oneshot_unit_handle_t` handle, `adc_channel_t` channel, const `adc_oneshot_chan_cfg_t` *config)

Set ADC oneshot mode required configurations.

**Note:** This API is thread-safe. For more details, see ADC programming guide

**Parameters**
- `handle` - [in] ADC handle
- `channel` - [in] ADC channel to be configured
- `config` - [in] ADC configurations

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments

`esp_err_t adc_oneshot_read` (`adc_oneshot_unit_handle_t` handle, `adc_channel_t` chan, int *out_raw)

Get one ADC conversion raw result.

**Note:** This API is thread-safe. For more details, see ADC programming guide

**Note:** This API should NOT be called in an ISR context
Chapter 2. API Reference

Parameters
• handle [in] ADC handle
• chan [in] ADC channel
• out_raw [out] ADC conversion raw result

Returns
• ESP_OK: On success
• ESP_ERR_INVALID_ARG: Invalid arguments
• ESP_ERR_TIMEOUT: Timeout, the ADC result is invalid

`esp_err_t adc_oneshot_del_unit(adc_oneshot_unit_handle_t handle)`
Delete the ADC unit handle.

Note: This API is thread-safe. For more details, see ADC programming guide

Parameters handle [in] ADC handle
Returns
• ESP_OK: On success
• ESP_ERR_INVALID_ARG: Invalid arguments
• ESP_ERR_NOT_FOUND: The ADC peripheral to be disclaimed isn’t in use

`esp_err_t adc_oneshot_io_to_channel(int io_num, adc_unit_t *const unit_id, adc_channel_t *const channel)`
Get ADC channel from the given GPIO number.

Parameters
• io_num [in] GPIO number
• unit_id [out] ADC unit
• channel [out] ADC channel
Returns
• ESP_OK: On success
• ESP_ERR_INVALID_ARG: Invalid argument
• ESP_ERR_NOT_FOUND: The IO is not a valid ADC pad

`esp_err_t adc_oneshot_channel_to_io(adc_unit_t unit_id, adc_channel_t channel, int *const io_num)`
Get GPIO number from the given ADC channel.

Parameters
• unit_id [in] ADC unit
• channel [in] ADC channel
• io_num [out] GPIO number
• – ESP_OK: On success
• – ESP_ERR_INVALID_ARG: Invalid argument

`esp_err_t adc_oneshot_get_calibrated_result(adc_oneshot_unit_handle_t handle, adc_cali_handle_t cali_handle, adc_channel_t chan, int *cali_result)`
Convenience function to get ADC calibrated result.

This is an all-in-one function which does:
• oneshot read ADC raw result
• calibrate the raw result and convert it into calibrated result (in mV)

Parameters
• handle [in] ADC oneshot handle, you should call adc_oneshot_new_unit() to get this handle
• cali_handle [in] ADC calibration handle, you should call adc_cali_create_scheme_x() in adc_cali_scheme.h to create a handle
• **chan [in]** ADC channel
• **cali_result [out]** Calibrated ADC result (in mV)

**Returns**
• ESP_OK Other return errors from adc_oneshot_read() and adc_cali_raw_to_voltage()

**Structures**

```c
struct adc_oneshot_unit_init_cfg_t
ADC oneshot driver initial configurations.

Public Members

adc_unit_t unit_id
ADC unit.

adc_oneshot_clk_src_t clk_src
Clock source.

adc_ulp_mode_t ulp_mode
ADC controlled by ULP, see adc_ulp_mode_t
```

```c
struct adc_oneshot_chan_cfg_t
ADC channel configurations.

Public Members

adc_atten_t atten
ADC attenuation.

adc_bitwidth_t bitwidth
ADC conversion result bits.
```

**Type Definitions**

typedef struct adc_oneshot_unit_ctx_t *adc_oneshot_unit_handle_t
Type of ADC unit handle for oneshot mode.

### 2.5.2 Analog to Digital Converter (ADC) Continuous Mode Driver

**Introduction**

The Analog to Digital Converter is integrated on the chip and is capable of measuring analog signals from specific analog IO pads. Additionally, the Direct Memory Access (DMA) functionality is utilized to efficiently retrieve ADC conversion results.

ESP32-S2 has two ADC unit(s), which can be used in scenarios like:

• Generate one-shot ADC conversion result
• Generate continuous ADC conversion results
This guide introduces ADC continuous mode conversion.

**Driver Concepts**  ADC continuous mode conversion is made up of multiple conversion frames.

- **Conversion Frame**: One conversion frame contains multiple conversion results. Conversion frame size is configured in `adc_continuous_new_handle()` in bytes.
- **Conversion Result**: One conversion result contains multiple bytes, see `SOC_ADC_DIGI_RESULT_BYTES`. Its structure is `adc_digi_output_data_t`, including ADC unit, ADC channel, and raw data.

**Functional Overview**

The following sections of this document cover the typical steps to install the ADC continuous mode driver, and read ADC conversion results from a group of ADC channels continuously:

- **Resource Allocation**: covers which parameters should be set up to initialize the ADC continuous mode driver and how to deinitialize it.
- **ADC Configurations**: describes how to configure the ADC(s) to make it work under continuous mode.
- **ADC Control**: describes ADC control functions.
- **Register Event Callbacks**: describes how to hook user-specific code to an ADC continuous mode event callback function.
- **Read Conversion Result**: covers how to get ADC conversion result.
- **Hardware Limitations**: describes the ADC-related hardware limitations.
- **IRAM Safe**: covers the IRAM safe functions.
- **Thread Safety**: lists which APIs are guaranteed to be thread-safe by the driver.

**Resource Allocation**  The ADC continuous mode driver is implemented based on ESP32-S2 SAR ADC module. Different ESP targets might have different numbers of independent ADCs.

To create an ADC continuous mode driver handle, set up the required configuration structure `adc_continuous_handle_cfg_t`:

- `adc_continuous_handle_cfg_t::max_store_buf_size`: set the maximum size of the pool in bytes, and the driver saves ADC conversion result into the pool. If this pool is full, new conversion results will be lost.
- `adc_continuous_handle_cfg_t::conv_frame_size`: set the size of the ADC conversion frame, in bytes.
- `adc_continuous_handle_cfg_t::flags`: set the flags that can change the driver’s behavior.
  - `flush_pool`: auto flush the pool when it’s full.

After setting up the above configurations for the ADC, call `adc_continuous_new_handle()` with the prepared `adc_continuous_handle_cfg_t`. This function may fail due to various errors such as invalid arguments, insufficient memory, etc.

Especially, when this function returns `ESP_ERR_NOT_FOUND`, this means the SPI3 peripheral is in use. See **Hardware Limitations** for more information.

If the ADC continuous mode driver is no longer used, you should deinitialize the driver by calling `adc_continuous_deinit()`.
Chapter 2. API Reference

**IIR filter** Two IIR filters are available when ADC is working in continuous mode. To create an ADC IIR filter, you should set up `adc_continuous_iir_filter_config_t` and call `adc_new_continuous_iir_filter()`.

- `adc_digi_filter_config_t::unit`: ADC unit.
- `adc_digi_filter_config_t::channel`: ADC channel to be filtered.
- `adc_digi_filter_config_t::coeff`: Filter coefficient.

On ESP32-S2, the filter is per ADC unit. Once a filter is enabled, all the enabled ADC channels in this ADC unit will be filtered. However, we suggest only enabling one ADC channel per unit, when using the filter feature. Because the filtered results depend on the previous filtered result. So you should not enable multiple ADC channels, to avoid mixing the filtered results.

To recycle a filter, you should call `adc_del_continuous_iir_filter()`.

**Monitor** 2 monitors are available when ADC is working under continuous mode, you can set one or two threshold(s) of a monitor on a working ADC channel, then the monitor will invoke interrupts every sample loop if conversion result outranges of the threshold. To create an ADC monitor, you need to set up the `adc_monitor_config_t` and call `adc_new_continuous_monitor()`.

- `adc_monitor_config_t::adc_unit`: Configures which ADC unit the channel you want to monitor belongs to.
- `adc_monitor_config_t::channel`: The channel you want to monitor.
- `adc_monitor_config_t::h_threshold`: The high threshold, conversion result larger than this value invokes interrupt, set to -1 if do not use.
- `adc_monitor_config_t::l_threshold`: The low threshold, conversion result less than this value invokes interrupt, set to -1 if do not use.

Once a monitor is created, you can operate it by following APIs to construct your apps.

- `adc_continuous_monitor_enable()`: Enable a monitor.
- `adc_continuous_monitor_disable()`: Disable a monitor.
- `adc_monitor_register_callbacks()`: register user callbacks to take action when the ADC value exceeds of the threshold.
- `adc_del_continuous_monitor()`: Delete a created monitor and free resources.

**Note:** There are some hardware limitations on ESP32-S2: 1. Only one threshold is supported for one monitor. 2. Only one monitor is supported for one ADC unit. 3. All enabled channel(s) of a certain ADC unit in ADC continuous mode driver will be monitored. The `adc_monitor_config_t::channel` parameter will not be used.

**Initialize the ADC Continuous Mode Driver**

```c
adc_continuous_handle_cfg_t adc_config = {
  .max_store_buf_size = 1024,
  .conv_frame_size = 100,
};
ESP_ERROR_CHECK(adc_continuous_new_handle(&adc_config));
```

**Recycle the ADC Unit**

```c
ESP_ERROR_CHECK(adc_continuous_deinit());
```

**ADC Configurations** After the ADC continuous mode driver is initialized, set up the `adc_continuous_config_t` to configure ADC IOs to measure analog signal:

- `adc_continuous_config_t::pattern_num`: number of ADC channels that will be used.
- `adc_continuous_config_t::adc_pattern`: list of configs for each ADC channel that will be used, see the description below.


- `adc_continuous_config_t::sample_freq_hz`: expected ADC sampling frequency in Hz.
- `adc_continuous_config_t::conv_mode`: continuous conversion mode.
- `adc_continuous_config_t::format`: conversion output format.

Set `adc_digi_pattern_config_t` with the following process:

- `adc_digi_pattern_config_t::atten`: ADC attenuation. Refer to the On-Chip Sensor and Analog Signal Processing chapter in TRM.
- `adc_digi_pattern_config_t::channel`: the IO corresponding ADC channel number. See the note below.
- `adc_digi_pattern_config_t::unit`: the ADC that the IO is subordinate to.
- `adc_digi_pattern_config_t::bit_width`: the bitwidth of the raw conversion result.

**Note:** For the IO corresponding ADC channel number, check TRM to acquire the ADC IOs. Besides, `adc_continuous_io_to_channel()` and `adc_continuous_channel_to_io()` can be used to acquire the ADC channels and ADC IOs.

To make these settings take effect, call `adc_continuous_config()` with the configuration structure above.

This API may fail due to reasons like `ESP_ERR_INVALID_ARG`. When it returns `ESP_ERR_INVALID_STATE`, this means the ADC continuous mode driver is started, you should not call this API at this moment.

See ADC continuous mode example `peripherals/adc/continuous_read` to see configuration codes.

To enable/disable the ADC IIR filter, you should call `adc_continuous_iir_filter_enable()` / `adc_continuous_iir_filter_disable()`.

To enable/disable the ADC monitor, you should call `adc_continuous_monitor_enable()` / `adc_continuous_monitor_disable()`.

**ADC Control**

**Start and Stop** Calling `adc_continuous_start()` makes the ADC start to measure analog signals from the configured ADC channels, and generate the conversion results.

On the contrary, calling `adc_continuous_stop()` stops the ADC conversion.

```
ESP_ERROR_CHECK(adc_continuous_stop());
```

**Register Event Callbacks** By calling `adc_continuous_register_event_callbacks()`, you can hook your own function to the driver ISR. Supported event callbacks are listed in `adc_continuous_evt_cbs_t`.

- `adc_continuous_evt_cbs_t::on_conv_done`: this is invoked when one conversion frame finishes.
- `adc_continuous_evt_cbs_t::on_pool_ovf`: this is invoked when the internal pool is full. Newer conversion results will be discarded.

As the above callbacks are called in an ISR context, you should always ensure the callback function is suitable for an ISR context. Blocking logic should not appear in these callbacks. The callback function prototype is declared in `adc_continuous_callback_t`.

You can also register your own context when calling `adc_continuous_register_event_callbacks()` by the parameter `user_data`. This user data will be passed to the callback functions directly.

This function may fail due to reasons like `ESP_ERR_INVALID_ARG`. Especially, when `CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE` is enabled, this error may indicate that the callback functions are not in the internal RAM. Check the error log for more details. Besides, when it fails due to `ESP_ERR_INVALID_STATE`, it indicates that the ADC continuous mode driver is started, and you should not add a callback at this moment.
**Conversion Done Event** When the driver completes a conversion, it triggers the `adc_continuous_evt_cbs_t::on_conv_done` event and fills the event data. Event data contains a buffer pointer to a conversion frame buffer, together with the size. Refer to `adc_continuous_evt_data_t` to know the event data structure.

**Note:** It is worth noting that, the data buffer `adc_continuous_evt_data_t::conv_frame_buffer` is maintained by the driver itself. Therefore, never free this piece of memory.

**Note:** When the Kconfig option `CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE` is enabled, the registered callbacks and the functions called by the callbacks should be placed in IRAM. The involved variables should be placed in internal RAM as well.

**Pool Overflow Event** The ADC continuous mode driver has an internal pool to save the conversion results. When the pool is full, a pool overflow event will emerge. Under this condition, the driver will not fill in the event data. This usually happens because the speed to read data from the pool by calling `adc_continuous_read()` is much slower than the ADC conversion speed.

**Read Conversion Result** After calling `adc_continuous_start()`, the ADC continuous conversion starts. Call `adc_continuous_read()` to get the conversion results of the ADC channels. You need to provide a buffer to get the raw results.

Function `adc_continuous_read()` tries to read the expected length of conversion results each time.

- When calling `adc_continuous_read()`, you can request to read a conversion result of the specified length. Sometimes, however, the actual available conversion results may be less than the requested length, in which case the function still moves the data from the internal pool into the buffer you provided. Therefore, to learn the number of conversion results actually moved into the buffer, please check the value of `out_length`.
- If there is no conversion result generated in the internal pool, the function will block for `timeout_ms` until the conversion results are generated. If there are still no generated results, the function will return `ESP_ERR_TIMEOUT`.
- If the generated results fill up the internal pool, newly generated results will be lost. Next time when `adc_continuous_read()` is called, this function will return `ESP_ERR_INVALID_STATE` to indicate this situation.

This API aims to give you a chance to read all the ADC continuous conversion results.

The ADC conversion results read from the above function are raw data. To calculate the voltage based on the ADC raw results, this formula can be used:

\[
V_{out} = D_{out} \times \frac{V_{max}}{D_{max}} \quad (1)
\]

where:

<table>
<thead>
<tr>
<th>Vout</th>
<th>Digital output result, standing for the voltage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dout</td>
<td>ADC raw digital reading result.</td>
</tr>
<tr>
<td>Vmax</td>
<td>Maximum measurable input analog voltage, this is related to the ADC attenuation, please refer to the On-Chip Sensor and Analog Signal Processing chapter in TRM.</td>
</tr>
<tr>
<td>Dmax</td>
<td>Maximum of the output ADC raw digital reading result, which is (2^\text{bitwidth}), where the bitwidth is the <code>adc_digi_pattern_config_t::bit_width</code> configured before.</td>
</tr>
</tbody>
</table>

To do further calibration to convert the ADC raw result to voltage in mV, please refer to *Analog to Digital Converter (ADC) Calibration Driver*. 

---

Espressif Systems

Submit Document Feedback

Release master
Hardware Limitations

- A specific ADC unit can only work under one operating mode at any one time, either continuous mode or one-shot mode. `adc_continuous_start()` has provided the protection.
- Random Number Generator (RNG) uses ADC as an input source. When ADC continuous mode driver works, the random number generated from RNG will be less random.
- ADC2 is also used by Wi-Fi. `adc_continuous_start()` has provided the protection between Wi-Fi driver and ADC continuous mode driver.
- ADC continuous mode driver uses SPI3 peripheral as hardware DMA FIFO. Therefore, if SPI3 is in use already, the `adc_continuous_new_handle()` will return ESP_ERR_NOT_FOUND.

Power Management

When power management is enabled, i.e., `CONFIG_PM_ENABLE` is on, the APB clock frequency may be adjusted when the system is in an idle state, thus potentially changing the behavior of ADC continuous conversion.

However, the continuous mode driver can prevent this change by acquiring a power management lock of type `ESP_PM_APB_FREQ_MAX`. The lock is acquired after the continuous conversion is started by `adc_continuous_start()`. Similarly, the lock will be released after `adc_continuous_stop()`. Therefore, `adc_continuous_start()` and `adc_continuous_stop()` should appear in pairs, otherwise, the power management will be out of action.

IRAM Safe

All the ADC continuous mode driver APIs are not IRAM-safe. They are not supposed to be run when the Cache is disabled. By enabling the Kconfig option `CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE`, the driver’s internal ISR handler is IRAM-safe, which means even when the Cache is disabled, the driver will still save the conversion results into its internal pool.

Thread Safety

ADC continuous mode driver APIs are not guaranteed to be thread-safe. However, the share hardware mutual exclusion is provided by the driver. See Hardware Limitations for more details.

Application Examples

- ADC continuous mode example: peripherals/adc/continuous_read.

API Reference

Header File

- `components/esp_adc/include/esp_adc/adc_continuous.h`
- This header file can be included with:

```c
#include "esp_adc/adc_continuous.h"
```
- This header file is a part of the API provided by the `esp_adc` component. To declare that your component depends on `esp_adc`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_adc
```

or

```cmake
PRIV_REQUIRES esp_adc
```

Functions

```c
esp_err_t adc_continuous_new_handle (const adc_continuous_handle_cfg_t *hdl_config,
                                        adc_continuous_handle_t *ret_handle)
```

Initialize ADC continuous driver and get a handle to it.
Chapter 2. API Reference

**Parameters**

- `hdl_config` - [in] Pointer to ADC initialization config. Refer to `adc_continuous_handle_cfg_t`.
- `ret_handle` - [out] ADC continuous mode driver handle

**Returns**

- ESP_ERR_INVALID_ARG If the combination of arguments is invalid.
- ESP_ERR_NOT_FOUND No free interrupt found with the specified flags
- ESP_ERR_NO_MEM If out of memory
- ESP_OK On success

```c
esp_err_t adc_continuous_config(adc_continuous_handle_t handle, const adc_continuous_config_t *config)
```

Set ADC continuous mode required configurations.

**Parameters**

- `handle` - [in] ADC continuous mode driver handle
- `config` - [in] Refer to `adc_digi_config_t`.

**Returns**

- ESP_ERR_INVALID_STATE: Driver state is invalid, you shouldn’t call this API at this moment
- ESP_ERR_INVALID_ARG: If the combination of arguments is invalid.
- ESP_OK: On success

```c
esp_err_t adc_continuous_register_event_callbacks(adc_continuous_handle_t handle, const adc_continuous_evt_cbs_t *cbs, void *user_data)
```

Register callbacks.

**Note:** User can deregister a previously registered callback by calling this function and setting the to-be-deregistered callback member in the `cbs` structure to NULL.

**Note:** When CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. Involved variables (including `user_data`) should be in internal RAM as well.

**Note:** You should only call this API when the ADC continuous mode driver isn’t started. Check return value to know this.

**Parameters**

- `handle` - [in] ADC continuous mode driver handle
- `cbs` - [in] Group of callback functions
- `user_data` - [in] User data, which will be delivered to the callback functions directly

**Returns**

- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_ERR_INVALID_STATE: Driver state is invalid, you shouldn’t call this API at this moment

```c
esp_err_t adc_continuous_start(adc_continuous_handle_t handle)
```

Start the ADC under continuous mode. After this, the hardware starts working.

**Parameters**

- `handle` - [in] ADC continuous mode driver handle

**Returns**

- ESP_ERR_INVALID_STATE: Driver state is invalid.
- ESP_OK On success
Chapter 2. API Reference

`esp_err_t adc_continuous_read(adc_continuous_handle_t handle, uint8_t* buf, uint32_t length_max, uint32_t* out_length, uint32_t timeout_ms)`

Read bytes from ADC under continuous mode.

**Parameters**
- `handle` - [in] ADC continuous mode driver handle
- `buf` - [out] Conversion result buffer to read from ADC. Suggest convert to `adc_digi_output_data_t` for ADC Conversion Results. See the subsection Driver Backgrounds in this header file to learn about this concept.
- `length_max` - [in] Expected length of the Conversion Results read from the ADC, in bytes.
- `out_length` - [out] Real length of the Conversion Results read from the ADC via this API, in bytes.
- `timeout_ms` - [in] Time to wait for data via this API, in millisecond.

**Returns**
- ESP_ERR_INVALID_STATE: Driver state is invalid. Usually it means the ADC sampling rate is faster than the task processing rate.
- ESP_ERR_TIMEOUT: Operation timed out
- ESP_OK: On success

`esp_err_t adc_continuous_stop(adc_continuous_handle_t handle)`

Stop the ADC. After this, the hardware stops working.

**Parameters**
- `handle` - [in] ADC continuous mode driver handle

**Returns**
- ESP_ERR_INVALID_STATE: Driver state is invalid.
- ESP_OK: On success

`esp_err_t adc_continuous_deinit(adc_continuous_handle_t handle)`

Deinitialize the ADC continuous driver.

**Parameters**
- `handle` - [in] ADC continuous mode driver handle

**Returns**
- ESP_ERR_INVALID_STATE: Driver state is invalid.
- ESP_OK: On success

`esp_err_t adc_continuous_flush_pool(adc_continuous_handle_t handle)`

Flush the driver internal pool.

**Note:** This API is not supposed to be called in an ISR context

**Parameters**
- `handle` - [in] ADC continuous mode driver handle

**Returns**
- ESP_ERR_INVALID_STATE: Driver state is invalid, you should call this API when it’s in init state
- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_OK: On success

`esp_err_t adc_continuous_io_to_channel(int io_num, adc_unit_t* const unit_id, adc_channel_t* const channel)`

Get ADC channel from the given GPIO number.

**Parameters**
- `io_num` - [in] GPIO number
- `unit_id` - [out] ADC unit
- `channel` - [out] ADC channel

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
• ESP_ERR_NOT_FOUND: The IO is not a valid ADC pad

`esp_err_t adc_continuous_channel_to_io(adc_unit_t unit_id, adc_channel_t channel, int *const io_num)`

Get GPIO number from the given ADC channel.

**Parameters**

- `unit_id` - [in] ADC unit
- `channel` - [in] ADC channel
- `io_num` - [out] GPIO number
- - ESP_OK: On success
- - ESP_ERR_INVALID_ARG: Invalid argument

**Structures**

`struct adc_continuous_handle_cfg_t`  
ADC continuous mode driver initial configurations.

**Public Members**

```c
uint32_t max_store_buf_size
```
Max length of the conversion results that driver can store, in bytes.

```c
uint32_t conv_frame_size
```
Conversion frame size, in bytes. This should be in multiples of
`SOC_ADC_DIGI_DATA_BYTES_PER_CONV`.

```c
uint32_t flush_pool
```
Flush the internal pool when the pool is full.

```c
struct adc_continuous_handle_cfg_t::[anonymous] flags
```
Driver flags.

`struct adc_continuous_config_t`  
ADC continuous mode driver configurations.

**Public Members**

```c
uint32_t pattern_num
```
Number of ADC channels that will be used.

```c
adc_digi_pattern_config_t *adc_pattern
```
List of configs for each ADC channel that will be used.

```c
uint32_t sample_freq_hz
```
The expected ADC sampling frequency in Hz. Please refer to `soc/soc_caps.h` to know available sampling frequency range.

```c
adc_digi_convert_mode_t conv_mode
```
ADC DMA conversion mode, see `adc_digi_convert_mode_t`.  

---

Espressif Systems  
Submit Document Feedback
adc_digi_output_format_t format

ADC DMA conversion output format, see adc_digi_output_format_t.

struct adc_continuous_evt_data_t

Event data structure.

Note: The conv_frame_buffer is maintained by the driver itself, so never free this piece of memory.

Public Members

uint8_t *conv_frame_buffer

Pointer to conversion result buffer for one conversion frame.

uint32_t size

Conversion frame size.

struct adc_continuous_evt_cbs_t

Group of ADC continuous mode callbacks.

Note: These callbacks are all running in an ISR environment.

Note: When CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. Involved variables should be in internal RAM as well.

Public Members

adc_continuous_callback_t on_conv_done

Event callback, invoked when one conversion frame is done. See the subsection Driver Backgrounds in this header file to learn about the conversion frame concept.

adc_continuous_callback_t on_pool_ovf

Event callback, invoked when the internal pool is full.

Macros

ADC_MAX_DELAY

ADC read max timeout value, it may make the adc_continuous_read block forever if the OS supports.

Type Definitions

typedef struct adc_continuous_ctx_t *adc_continuous_handle_t

Type of adc continuous mode driver handle.
Prototype of ADC continuous mode event callback.

**Param handle** [in] ADC continuous mode driver handle  
**Param edata** [in] Pointer to ADC continuous mode event data  
**Param user_data** [in] User registered context, registered when in `adc_continuous_register_event_callbacks()`  
**Return** Whether a high priority task is woken up by this function

### 2.5.3 Analog to Digital Converter (ADC) Calibration Driver

#### Introduction

In ESP32-S2, the digital-to-analog converter (ADC) compares the input analog voltage to the reference, and determines each bit of the output digital result. By design, the ADC reference voltage for ESP32-S2 is 1100 mV. However, the true reference voltage can range from 1000 mV to 1200 mV among different chips. This guide introduces the ADC calibration driver to minimize the effect of different reference voltages, and get more accurate output results.

#### Functional Overview

The following sections of this document cover the typical steps to install and use the ADC calibration driver:

- **Calibration Scheme Creation** - covers how to create a calibration scheme handle and delete the calibration scheme handle.
- **Result Conversion** - covers how to convert ADC raw result to calibrated result.
- **Thread Safety** - lists which APIs are guaranteed to be thread-safe by the driver.
- **Minimize Noise** - describes a general way to minimize the noise.

#### Calibration Scheme Creation

The ADC calibration driver provides ADC calibration scheme(s). From the calibration driver’s point of view, an ADC calibration scheme is created for an ADC calibration handle `adc_cali_handle_t`.  

The function `adc_cali_check_scheme()` can be used to know which calibration scheme is supported on the chip. If you already know the supported schemes, this step can be skipped. Just call the corresponding function to create the scheme handle.

If you use your custom ADC calibration schemes, you could either modify this function `adc_cali_check_scheme()`, or just skip this step and call your custom creation function.

#### ADC Calibration Line Fitting Scheme

ESP32-S2 supports `ADC_CALI_SCHEME_VER_LINE_FITTING` scheme. To create this scheme, set up `adc_cali_line_fitting_config_t` first.

- `adc_cali_line_fitting_config_t::unit_id`, the ADC that your ADC raw results are from.  
- `adc_cali_line_fitting_config_t::atten`, ADC attenuation that your ADC raw results use.  
- `adc_cali_line_fitting_config_t::bitwidth`, bit width of ADC raw result.

After setting up the configuration structure, call `adc_cali_create_scheme_line_fitting()` to create a Line Fitting calibration scheme handle.

This function may fail due to reasons such as `ESP_ERR_INVALID_ARG` or `ESP_ERR_NO_MEM`. Especially, when the function returns `ESP_ERR_NOT_SUPPORTED`, this means the calibration scheme required eFuse bits are not burned on your board.
When the ADC calibration is no longer used, please delete the calibration scheme handle by calling `adc_cali_delete_scheme_line_fitting()`.

**Delete Line Fitting Scheme**

```c
ESP_LOGI(TAG, "delete %s calibration scheme", "Line Fitting");
ESP_ERROR_CHECK(adc_cali_delete_scheme_line_fitting(handle));
```

**Note**: If you want to use your custom calibration schemes, you could provide a creation function to create your calibration scheme handle. Check the function table `adc_cali_scheme_t` in `components/esp_adc/interface/adc_cali_interface.h` to know the ESP ADC calibration interface.

**Result Conversion** After setting up the calibration characteristics, you can call `adc_cali_raw_to_voltage()` to convert the ADC raw result into calibrated result. The calibrated result is in the unit of mV. This function may fail due to an invalid argument. Especially, if this function returns `ESP_ERR_INVALID_STATE`, this means the calibration scheme is not created. You need to create a calibration scheme handle, use `adc_cali_check_scheme()` to know the supported calibration scheme. On the other hand, you could also provide a custom calibration scheme and create the handle.

**Get Voltage**

```c
ESP_ERROR_CHECK(adc_cali_raw_to_voltage(adc_cali_handle, adc_raw[0][0], &voltage[0][0]));
ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV", ADC_UNIT_1 + 1, EXAMPLE_ --ADC1_CHAN0, voltage[0][0]);
```

**Thread Safety** The factory function `esp_adc_cali_new_scheme()` is guaranteed to be thread-safe by the driver. Therefore, you can call them from different RTOS tasks without protection by extra locks. Other functions that take the `adc_cali_handle_t` as the first positional parameter are not thread-safe, you should avoid calling them from multiple tasks.

**Minimize Noise** The ESP32-S2 ADC is sensitive to noise, leading to large discrepancies in ADC readings. Depending on the usage scenario, you may need to connect a bypass capacitor (e.g., a 100 nF ceramic capacitor) to the ADC input pad in use, to minimize noise. Besides, multisampling may also be used to further mitigate the effects of noise.

**API Reference**

**Header File**

- `components/esp_adc/include/esp_adc/adc_cali.h`
- This header file can be included with:

```c
#include "esp_adc/adc_cali.h"
```
This header file is a part of the API provided by the `esp_adc` component. To declare that your component depends on `esp_adc`, add the following to your `CMakeLists.txt`:

```cmake
REQUIRES esp_adc
```

or

```cmake
PRIV_REQUIRES esp_adc
```

### Functions

**`esp_err_t adc_cali_check_scheme`**

Type: `adc_cali_scheme_ver_t` *scheme_mask*

Check the supported ADC calibration scheme.

**Parameters**
- `scheme_mask` - [out] Supported ADC calibration scheme(s)

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NOT_SUPPORTED: No supported calibration scheme

**`esp_err_t adc_cali_raw_to_voltage`**

Convert ADC raw data to calibrated voltage.

**Parameters**
- `handle` - [in] ADC calibration handle
- `raw` - [in] ADC raw data
- `voltage` - [out] Calibrated ADC voltage (in mV)

**Returns**
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_INVALID_STATE: Invalid state, scheme didn’t register

### Type Definitions

typedef struct adc_cali_scheme_t *`adc_cali_handle_t`

ADC calibration handle.

### Enumerations

enum `adc_cali_scheme_ver_t`

ADC calibration scheme.

**Values**:

- `ADC_CALI_SCHEME_VER_LINE_FITTING` Line fitting scheme.
- `ADC_CALI_SCHEME_VER_CURVE_FITTING` Curve fitting scheme.

### Header File

- components/esp_adc/include/esp_adc/adc_cali_scheme.h

This header file can be included with:

```
#include "esp_adc/adc_cali_scheme.h"
```

This header file is a part of the API provided by the `esp_adc` component. To declare that your component depends on `esp_adc`, add the following to your `CMakeLists.txt`:

```cmake
REQUIRES esp_adc
```

or

```cmake
PRIV_REQUIRES esp_adc
```
2.5.4 Clock Tree

The clock subsystem of ESP32-S2 is used to source and distribute system/module clocks from a range of root clocks. The clock tree driver maintains the basic functionality of the system clock and the intricate relationship among module clocks.

This document starts with the introduction to root and module clocks. Then it covers the clock tree APIs that can be called to monitor the status of the module clocks at runtime.

Introduction

This section lists definitions of ESP32-S2’s supported root clocks and module clocks. These definitions are commonly used in the driver configuration, to help select a proper source clock for the peripheral.

Root Clocks Root clocks generate reliable clock signals. These clock signals then pass through various gates, muxes, dividers, or multipliers to become the clock sources for every functional module: the CPU core(s), Wi-Fi, Bluetooth, the RTC, and the peripherals.

ESP32-S2’s root clocks are listed in \texttt{soc\_root\_clk\_t}:

- **Internal 8 MHz RC Oscillator (RC\_FAST)**
  
  This RC oscillator generates a about 8.5 MHz clock signal output as the \texttt{RC\_FAST\_CLK}.
  The about 8.5 MHz signal output is also passed into a configurable divider, which by default divides the input clock frequency by 256, to generate a \texttt{RC\_FAST\_D256\_CLK}.
  The exact frequency of \texttt{RC\_FAST\_CLK} can be computed in runtime through calibration on the \texttt{RC\_FAST\_D256\_CLK}.

- **External 40 MHz Crystal (XTAL)**

- **Internal 90 kHz RC Oscillator (RC\_SLOW)**

  This RC oscillator generates a about 90kHz clock signal output as the \texttt{RC\_SLOW\_CLK}.
  The exact frequency of this clock can be computed in runtime through calibration.

- **External 32 kHz Crystal - optional (XTAL32K)**

  The clock source for this \texttt{XTAL32K\_CLK} can be either a 32 kHz crystal connecting to the \texttt{XTAL\_32K\_P} and \texttt{XTAL\_32K\_N} pins or a 32 kHz clock signal generated by an external circuit. The external signal must be connected to the \texttt{XTAL\_32K\_P} pin.
  \texttt{XTAL32K\_CLK} can also be calibrated to get its exact frequency.

Typically, the frequency of the signal generated from an RC oscillator circuit is less accurate and more sensitive to the environment compared to the signal generated from a crystal. ESP32-S2 provides several clock source options for the \texttt{RTC\_SLOW\_CLK}, and it is possible to make the choice based on the requirements for system time accuracy and power consumption. For more details, please refer to \textit{RTC Timer Clock Sources}.

Module Clocks ESP32-S2’s available module clocks are listed in \texttt{soc\_module\_clk\_t}. Each module clock has a unique ID. You can get more information on each clock by checking the documented enum value.
API Usage

The clock tree driver provides an all-in-one API to get the frequency of the module clocks, `esp_clk_tree_src_get_freq_hz()`. This function allows you to obtain the clock frequency at any time by providing the clock name `soc_module_clk_t` and specifying the desired precision level for the returned frequency value `esp_clk_tree_src_freq_precision_t`.

API Reference

Header File

- components/soc/esp32s2/include/soc/clk_tree_defs.h
- This header file can be included with:

```
#include "soc/clk_tree_defs.h"
```

Macros

- `SOC_CLK_RC_FAST_FREQ_APPROX`: Approximate RC_FAST_CLK frequency in Hz
- `SOC_CLK_RC_SLOW_FREQ_APPROX`: Approximate RC_SLOW_CLK frequency in Hz
- `SOC_CLK_RC_FAST_D256_FREQ_APPROX`: Approximate RC_FAST_D256_CLK frequency in Hz
- `SOC_CLK_XTAL32K_FREQ_APPROX`: Approximate XTAL32K_CLK frequency in Hz
- `SOC_GPTIMER_CLKS`: Array initializer for all supported clock sources of GPTimer.
  The following code can be used to iterate all possible clocks:
  ```
soc_periph_gptimer_clk_src_t gptimer_clks[] = (soc_periph_gptimer_clk_src_t)SOC_GPTIMER_CLKS;
for (size_t i = 0; i < sizeof(gptimer_clks) / sizeof(gptimer_clks[0]); i++) {
    soc_periph_gptimer_clk_src_t clk = gptimer_clks[i];
    // Test GPTimer with the clock 'clk'
}
```
- `SOC_LCD_CLKS`: Array initializer for all supported clock sources of LCD.
- `SOC_RMT_CLKS`: Array initializer for all supported clock sources of RMT.
- `SOC_TEMP_SENSOR_CLKS`: Array initializer for all supported clock sources of Temperature Sensor.
- `SOC_I2S_CLKS`: Array initializer for all supported clock sources of I2S.
Chapter 2. API Reference

SOC_I2C_CLKS
Array initializer for all supported clock sources of I2C.

SOC_SPI_CLKS
Array initializer for all supported clock sources of SPI.

SOC_SDM_CLKS
Array initializer for all supported clock sources of SDM.

SOC_GLITCH_FILTER_CLKS
Array initializer for all supported clock sources of Glitch Filter.

SOC_DAC_DIGI_CLKS
Array initializer for all supported clock sources of DAC digital controller.

SOC_DAC_COSINE_CLKS
Array initializer for all supported clock sources of DAC cosine wave generator.

SOC_TWAI_CLKS
Array initializer for all supported clock sources of TWAI.

SOC_ADC_DIGI_CLKS
Array initializer for all supported clock sources of ADC digital controller.

SOC_ADC_RTC_CLKS
Array initializer for all supported clock sources of ADC RTC controller.

SOC_MWDT_CLKS
Array initializer for all supported clock sources of MWDT.

SOC_LEDC_CLKS
Array initializer for all supported clock sources of LEDC.

Enumerations

enum soc_root_clk_t
Root clock.

Values:

enumerator SOC_ROOT_CLK_INT_RC_FAST
Internal 8MHz RC oscillator

enumerator SOC_ROOT_CLK_INT_RC_SLOW
Internal 90kHz RC oscillator

enumerator SOC_ROOT_CLK_EXT_XTAL
External 40MHz crystal
Chapter 2. API Reference

enumerator SOC_ROOT_CLK_EXT_XTAL32K
    External 32kHz crystal/clock signal

enum soc_cpu_clk_src_t
    CPU_CLK mux inputs, which are the supported clock sources for the CPU_CLK.

    Note: Enum values are matched with the register field values on purpose

    Values:

    enumerator SOC_CPU_CLK_SRC_XTAL
        Select XTAL_CLK as CPU_CLK source

    enumerator SOC_CPU_CLK_SRC_PLL
        Select PLL_CLK as CPU_CLK source (PLL_CLK is the output of 40MHz crystal oscillator frequency
        multiplier, can be 480MHz or 320MHz)

    enumerator SOC_CPU_CLK_SRC_RC_FAST
        Select RC_FAST_CLK as CPU_CLK source

    enumerator SOC_CPU_CLK_SRC_APLL
        Select APLL_CLK as CPU_CLK source

    enumerator SOC_CPU_CLK_SRC_INVALID
        Invalid CPU_CLK source

enum soc_rtc_slow_clk_src_t
    RTC_SLOW_CLK mux inputs, which are the supported clock sources for the RTC_SLOW_CLK.

    Note: Enum values are matched with the register field values on purpose

    Values:

    enumerator SOC_RTC_SLOW_CLK_SRC_RC_SLOW
        Select RC_SLOW_CLK as RTC_SLOW_CLK source

    enumerator SOC_RTC_SLOW_CLK_SRC_XTAL32K
        Select XTAL32K_CLK as RTC_SLOW_CLK source

    enumerator SOC_RTC_SLOW_CLK_SRC_RC_FAST_D256
        Select RC_FAST_D256_CLK (referred as FOSC_DIV or 8m_d256/8md256 in TRM and reg. description) as RTC_SLOW_CLK source

    enumerator SOC_RTC_SLOW_CLK_SRC_INVALID
        Invalid RTC_SLOW_CLK source

enum soc_rtc_fast_clk_src_t
    RTC_FAST_CLK mux inputs, which are the supported clock sources for the RTC_FAST_CLK.
**Note:** Enum values are matched with the register field values on purpose

**Values:**

enumerator `SOC_RTC_FAST_CLK_SRC_XTAL_D4`
Select XTAL_D4_CLK (may referred as XTAL_CLK_DIV_4) as RTC_FAST_CLK source

enumerator `SOC_RTC_FAST_CLK_SRC_XTAL_DIV`
Alias name for `SOC_RTC_FAST_CLK_SRC_XTAL_D4`

enumerator `SOC_RTC_FAST_CLK_SRC_RC_FAST`
Select RC_FAST_CLK as RTC_FAST_CLK source

enumerator `SOC_RTC_FAST_CLK_SRC_INVALID`
Invalid RTC_FAST_CLK source

**Note:** Enum starts from 1, to save 0 for special purpose

**Values:**

enumerator `SOC_MOD_CLK_CPU`
CPU_CLK can be sourced from XTAL, PLL, RC_FAST, or APLL by configuring `soc_cpu_clk_src_t`

enumerator `SOC_MOD_CLK_RTC_FAST`
RTC_FAST_CLK can be sourced from XTAL_D4 or RC_FAST by configuring `soc_rtc_fast_clk_src_t`

enumerator `SOC_MOD_CLK_RTC_SLOW`
RTC_SLOW_CLK can be sourced from RC_SLOW, XTAL32K, or RC_FAST_D256 by configuring `soc_rtc_slow_clk_src_t`

enumerator `SOC_MOD_CLK_APB`
APB_CLK is highly dependent on the CPU_CLK source

enumerator `SOC_MOD_CLK_PLL_F160M`
PLL_F160M_CLK is derived from PLL, and has a fixed frequency of 160MHz

enumerator `SOC_MOD_CLK_XTAL32K`
XTAL32K_CLK comes from the external 32kHz crystal, passing a clock gating to the peripherals

enumerator `SOC_MOD_CLK_RC_FAST`
RC_FAST_CLK comes from the internal 8MHz rc oscillator, passing a clock gating to the peripherals

enumerator `SOC_MOD_CLK_RC_FAST_D256`
RC_FAST_D256_CLK is derived from the internal 8MHz rc oscillator, divided by 256, and passing a clock gating to the peripherals
Chapter 2. API Reference

enumerator **SOC_MOD_CLK_XTAL**

XTAL_CLK comes from the external 40MHz crystal

enumerator **SOC_MOD_CLK_REF_TICK**

REF_TICK is derived from XTAL or RC_FAST via a divider, it has a fixed frequency of 1MHz by default

enumerator **SOC_MOD_CLK_APLL**

APLL is sourced from PLL, and its frequency is configurable through APLL configuration registers

enumerator **SOC_MOD_CLK_TEMP_SENSOR**

TEMP_SENSOR_CLK comes directly from the internal 8MHz rc oscillator

enumerator **SOC_MOD_CLK_INVALID**

Indication of the end of the available module clock sources

enum **soc_periph_systimer_clk_src_t**

Type of SYSTIMER clock source.

Values:

enumerator **SYSTIMER_CLK_SRC_XTAL**

SYSTIMER source clock is XTAL

enumerator **SYSTIMER_CLK_SRC_DEFAULT**

SYSTIMER source clock default choice is XTAL

enum **soc_periph_gptimer_clk_src_t**

Type of GPTimer clock source.

Values:

enumerator **GPTIMER_CLK_SRC_APB**

Select APB as the source clock

enumerator **GPTIMER_CLK_SRC_XTAL**

Select XTAL as the source clock

enumerator **GPTIMER_CLK_SRC_DEFAULT**

Select APB as the default choice

enum **soc_periph_tg_clk_src_legacy_t**

Type of Timer Group clock source, reserved for the legacy timer group driver.

Values:

enumerator **TIMER_SRC_CLK_APB**

Timer group source clock is APB

enumerator **TIMER_SRC_CLK_XTAL**

Timer group source clock is XTAL
enumerator **TIMER_SRC_CLK_DEFAULT**

Timer group source clock default choice is APB

enum **soc_periph_lcd_clk_src_t**

Type of LCD clock source.

*Values:*

enumerator **LCD_CLK_SRC_PLL160M**

Select PLL_F160M as the source clock

enumerator **LCD_CLK_SRC_DEFAULT**

Select PLL_F160M as the default choice

enum **soc_periph_rmt_clk_src_t**

Type of RMT clock source.

*Values:*

enumerator **RMT_CLK_SRC_APB**

Select APB as the source clock

enumerator **RMT_CLK_SRC_REF_TICK**

Select REF_TICK as the source clock

enumerator **RMT_CLK_SRC_DEFAULT**

Select APB as the default choice

enum **soc_periph_rmt_clk_src_legacy_t**

Type of RMT clock source, reserved for the legacy RMT driver.

*Values:*

enumerator **RMT_BASECLK_APB**

RMT source clock is APB CLK

enumerator **RMT_BASECLK_REF**

RMT source clock is REF_TICK

enumerator **RMT_BASECLK_DEFAULT**

RMT source clock default choice is APB

enum **soc_periph_temperature_sensor_clk_src_t**

Type of Temp Sensor clock source.

*Values:*

enumerator **TEMPERATURE_SENSOR_CLK_SRC_RC_FAST**

Select RC_FAST as the source clock

enumerator **TEMPERATURE_SENSOR_CLK_SRC_DEFAULT**

Select RC_FAST as the default choice
enum `soc_periph_uart_clk_src_legacy_t`
Type of UART clock source, reserved for the legacy UART driver.

*Values:*

enumerator `UART_SCLK_APB`
UART source clock is APB CLK

enumerator `UART_SCLK_REF_TICK`
UART source clock is REF_TICK

enumerator `UART_SCLK_DEFAULT`
UART source clock default choice is APB

enum `soc_periph_i2s_clk_src_t`
I2S clock source enum.

*Values:*

enumerator `I2S_CLK_SRC_DEFAULT`
Select PLL_F160M as the default source clock

enumerator `I2S_CLK_SRC_PLL_160M`
Select PLL_F160M as the source clock

enumerator `I2S_CLK_SRC_APLL`
Select APLL as the source clock

enum `soc_periph_i2c_clk_src_t`
Type of I2C clock source.

*Values:*

enumerator `I2C_CLK_SRC_APB`

enumerator `I2C_CLK_SRC_REF_TICK`

enumerator `I2C_CLK_SRC_DEFAULT`

enum `soc_periph_spi_clk_src_t`
Type of SPI clock source.

*Values:*

enumerator `SPI_CLK_SRC_DEFAULT`
Select APB as SPI source clock

enumerator `SPI_CLK_SRC_APB`
Select XTAL as SPI source clock
Chapter 2. API Reference

```c
enum soc_periph_sdm_clk_src_t
Sigma Delta Modulator clock source.
Values:

enumerator SDM_CLK_SRC_APB
Select APB as the source clock

enumerator SDM_CLK_SRC_DEFAULT
Select APB as the default clock choice

enum soc_periph_glitch_filter_clk_src_t
Glitch filter clock source.
Values:

enumerator GLITCH_FILTER_CLK_SRC_APB
Select APB clock as the source clock

enumerator GLITCH_FILTER_CLK_SRC_DEFAULT
Select APB clock as the default clock choice

enum soc_periph_dac_digi_clk_src_t
DAC digital controller clock source.
Values:

enumerator DAC_DIGI_CLK_SRC_APB
Select APB as the source clock

enumerator DAC_DIGI_CLK_SRC_APLL
Select APLL as the source clock

enumerator DAC_DIGI_CLK_SRC_DEFAULT
Select APB as the default source clock

enum soc_periph_dac_cosine_clk_src_t
DAC cosine wave generator clock source.
Values:

enumerator DAC_COSINE_CLK_SRC_RTC_FAST
Select RTC FAST as the source clock

enumerator DAC_COSINE_CLK_SRC_DEFAULT
Select RTC FAST as the default source clock

enum soc_periph_twai_clk_src_t
TWAI clock source.
Values:
```
enumerator TWAI_CLK_SRC_APB
Select APB as the source clock

enumerator TWAI_CLK_SRC_DEFAULT
Select APB as the default clock choice

enum soc_periph_adc_digi_clk_src_t
ADC digital controller clock source.
Values:

enumerator ADC_DIGI_CLK_SRC_APB
Select APB as the source clock

enumerator ADC_DIGI_CLK_SRC_APLL
Select APLL as the source clock

enumerator ADC_DIGI_CLK_SRC_DEFAULT
Select APB as the default clock choice

enum soc_periph_adc_rtc_clk_src_t
ADC RTC controller clock source.
Values:

enumerator ADC_RTC_CLK_SRC_RC_FAST
Select RC_FAST as the source clock

enumerator ADC_RTC_CLK_SRC_DEFAULT
Select RC_FAST as the default clock choice

enum soc_periph_mwdt_clk_src_t
MWDT clock source.
Values:

enumerator MWDT_CLK_SRC_APB
Select APB as the source clock

enumerator MWDT_CLK_SRC_DEFAULT
Select APB as the default clock choice

enum soc_periph_ledc_clk_src_legacy_t
Type of LEDC clock source, reserved for the legacy LEDC driver.
Values:

enumerator LEDC_AUTO_CLK
LEDC source clock will be automatically selected based on the giving resolution and duty parameter when init the timer
enumerator LEDC_USE_APB_CLK
Select APB as the source clock

denumerator LEDC_USE_RC_FAST_CLK
Select RC_FAST as the source clock

denumerator LEDC_USE_REF_TICK
Select REF_TICK as the source clock

denumerator LEDC_USE_XTAL_CLK
Select XTAL as the source clock

denumerator LEDC_USE_RTC8M_CLK
Alias of ‘LEDC_USE_RC_FAST_CLK’

enum soc_clkout_sig_id_t
Values:

denumerator CLKOUT_SIG_PLL
PLL_CLK is the output of crystal oscillator frequency multiplier

denumerator CLKOUT_SIG_RC_SLOW
RC slow clock, depends on the RTC_CLK_SRC configuration

denumerator CLKOUT_SIG_XTAL
Main crystal oscillator clock

denumerator CLKOUT_SIG_APLL
Divided by PLL, frequency is configurable

denumerator CLKOUT_SIG_REF_TICK
Divided by APB clock, usually be 1MHz

denumerator CLKOUT_SIG_PLL_F80M
From PLL, usually be 80MHz

denumerator CLKOUT_SIG_RC_FAST
RC fast clock, about 17.5MHz

denumerator CLKOUT_SIG_INVALID

Header File

- components/esp_hw_support/include/esp_clk_tree.h
- This header file can be included with:

```
#include "esp_clk_tree.h"
```
Get frequency of module clock source.

**Parameters**

- **clk_src** - [in] Clock source available to modules, in `soc_module_clk_t`
- **precision** - [in] Degree of precision, one of `esp_clk_tree_src_freq_precision_t` values. This arg only applies to the clock sources that their frequencies can vary: `SOC_MOD_CLK_RTC_FAST`, `SOC_MOD_CLK_RTC_SLOW`, `SOC_MOD_CLK_RC_FAST`, `SOC_MOD_CLK_RC_FAST_D256`, `SOC_MOD_CLK_XTAL32K`. For other clock sources, this field is ignored.
- **freq_value** - [out] Frequency of the clock source, in Hz

**Returns**

- **ESP_OK** Success
- **ESP_ERR_INVALID_ARG** Parameter error
- **ESP_FAIL** Calibration failed

**Enumerations**

```cpp
eenum esp_clk_tree_src_freq_precision_t

Degree of precision of frequency value to be returned by `esp_clk_tree_src_get_freq_hz()`

*ESP_CLK_TREE_SRC_FREQ_PRECISION_CACHED*

*ESP_CLK_TREE_SRC_FREQ_PRECISION_APPROX*

*ESP_CLK_TREE_SRC_FREQ_PRECISION_EXACT*

*ESP_CLK_TREE_SRC_FREQ_PRECISION_INVALID*
```

### 2.5.5 Digital To Analog Converter (DAC)

**Overview**

ESP32-S2 has two 8-bit DAC (digital to analog converter) channels respectively connected to GPIO17 (Channel 1) and GPIO18 (Channel 2). Each DAC channel can convert the digital value 0~255 to the analog voltage 0~Vref (The reference voltage Vref here is input from the pin VDD3P3_RTC_IO, which ideally equals to the power supply VDD). The output voltage can be calculated as the following:

\[
\text{out\_voltage} = \frac{\text{Vref}}{255} \times \text{digi\_val}
\]

The DAC peripheral supports outputting analog signal in the following ways:

1. Outputting a voltage directly. The DAC channel keeps outputting a specified voltage.
2. Outputting continuous analog signal by DMA. The DAC converts the data in a buffer at a specified frequency.
3. Outputting a cosine wave by the cosine wave generator. The DAC channel can output a cosine wave with specified frequency and amplitude.

For other analog output options, see **Sigma-Delta Modulation** and **LED Control**. Both modules produce high-frequency PWM/PDM output, which can be hardware low-pass filtered in order to generate a lower frequency analog output.
Public headers that need to be included in the DAC application are listed as follows:

- **dac.h**: The top header file of the legacy DAC driver, which should be only included in the apps which use the legacy driver API.
- **dac_oneshot.h**: The top header file of the new DAC driver, which should be included in the apps which use the new driver API with one-shot mode.
- **dac_cosine.h**: The top header file of the new DAC driver, which should be included in the apps which use the new driver API with cosine mode.
- **dac_continuous.h**: The top header file of the new DAC driver, which should be included in the apps which use the new driver API with continuous mode.

**Note**: The legacy driver cannot coexist with the new driver. Include **dac.h** to use the legacy driver or **dac_oneshot.h**, **dac_cosine.h**, and **dac_continuous.h** to use the new driver. The legacy driver might be removed in the future.

---

**Functional Overview**

**Resources Management** The DAC on ESP32-S2 has two channels. The channels have separate software resources and can be managed by **dac_oneshot_handle_t**, **dac_cosine_handle_t**, or **dac_continuous_handle_t** according to the usage. Registering different modes on a same DAC channel is not allowed.

**Direct Voltage Output (One-shot/Direct Mode)** The DAC channels in the group can convert an 8-bit digital value into the analog when **dac_oneshot_output_voltage()** is called (it can be called in ISR). The analog voltage is kept on the DAC channel until the next conversion starts. To start the voltage conversion, the DAC channels need to be enabled first through registering by **dac_oneshot_new_channel()**.

**Continuous Wave Output (Continuous/DMA Mode)** DAC channels can convert digital data continuously via the DMA. There are three ways to write the DAC data:

1. Normal writing (synchronous): Data can be transmitted at one time and kept blocked until all the data has been loaded into the DMA buffer, and the voltage is kept as the last conversion value while no more data is
inputted. It is usually used to transport a long signal like an audio. To convert data continuously, the continuous channel handle need to be allocated by calling `dac_continuous_new_channels()` and the DMA conversion should be enabled by calling `dac_continuous_enable()`. Then data can be written by `dac_continuous_write()` synchronously. Refer to `peripherals/dac/dac_continuous/dac_audio` for examples.

2. Cyclical writing: A piece of data can be converted cyclically without blocking, and no more operation is needed after the data are loaded into the DMA buffer. But note that the inputted buffer size is limited by the number of descriptors and the DMA buffer size. It is usually used to transport short signals that need to be repeated, e.g., a sine wave. To achieve cyclical writing, call `dac_continuous_write_cyclically()` after the DAC continuous mode is enabled. Refer to `peripherals/dac/dac_continuous/signal_generator` for examples.

3. Asynchronous writing: Data can be transmitted asynchronously based on the event callback. `dac_event_callbacks_t::on_convert_done` must be registered to use asynchronous mode. Users can get the `dac_event_data_t` in the callback which contains the DMA buffer address and length, allowing them to load the data into the buffer directly. To use the asynchronous writing, call `dac_continuous_register_event_callback()` to register the `dac_event_callbacks_t::on_convert_done` before enabling, and then `dac_continuous_start_async_writing()` to start the asynchronous writing. Note that once the asynchronous writing is started, the callback function will be triggered continuously. Call `dac_continuous_write_asynchronously()` to load the data either in a separate task or in the callback directly. Refer to `peripherals/dac/dac_continuous/dac_audio` for examples.

On ESP32-S2, the DAC digital controller can be connected internally to the SPI3 and use its DMA for continuous conversion.

The clock sources of the DAC digital controller include:

- `dac_continuous_digi_clk_src_t::DAC_DIGI_CLK_SRC_APB` supports frequency between 77 Hz to several MHz. It is the default clock which can also be selected by `dac_continuous_digi_clk_src_t::DAC_DIGI_CLK_SRC_DEFAULT`.
- `dac_continuous_digi_clk_src_t::DAC_DIGI_CLK_SRC_APLL` supports frequency between 6 Hz to several MHz. However, it might be occupied by other peripherals, thus not providing the required frequency. In such case, this clock source is available only if APLL still can be correctly divided into the target DAC DMA frequency.

Cosine Wave Output (Cosine Mode) The DAC peripheral has a cosine wave generator, which can generate cosine wave on the channels. Users can specify the frequency, amplitude, and phase of the cosine wave. To output the cosine wave, please acquire the DAC to cosine mode using `dac_cosine_new_channel()`, and then start the cosine wave generator by `dac_cosine_start()`.

Currently, the clock source of the cosine wave generator only comes from `RTC_FAST` which can be selected by `dac_cosine_clk_src_t::DAC_COSINE_CLK_SRC_RTC_FAST`. It is also the default clock source which is the same as `dac_cosine_clk_src_t::DAC_COSINE_CLK_SRC_RTC_DEFAULT`.

Power Management When the power management is enabled (i.e., `CONFIG_PM_ENABLE` is on), the system will adjust or stop the clock source of DAC before entering Light-sleep mode, thus potential influence to the DAC signals may lead to false data conversion.

When using DAC driver in continuous mode, it can prevent the system from changing or stopping the clock source in DMA or cosine mode by acquiring a power management lock. When the clock source is generated from APB, the lock type will be set to `esp_pm_lock_type_t::ESP_PM_APB_FREQ_MAX`. When the clock source is APLL (only in DMA mode), it will be set to `esp_pm_lock_type_t::ESP_PM_NO_LIGHT_SLEEP`. Whenever the DAC is converting (i.e., DMA or cosine wave generator is working), the driver guarantees that the power management lock is acquired after calling `dac_continuous_enable()`. Likewise, the driver will release the lock when `dac_continuous_disable()` is called.

IRAM Safe By default, the DAC DMA interrupt will be deferred when the cache is disabled for reasons like writing/erasing Flash. Thus the DMA EOF interrupt will not get executed in time.
To avoid such case in real-time applications, you can enable the Kconfig option `CONFIG_DAC_ISR_IRAM_SAFE` which:

1. Enables the interrupt being serviced even when cache is disabled;
2. Places driver object into DRAM (in case it is linked to PSRAM by accident).

This allows the interrupt to run while the cache is disabled but comes at the cost of increased IRAM consumption.

**Thread Safety** All the public DAC APIs are guaranteed to be thread safe by the driver, which means users can call them from different RTOS tasks without protection by extra locks. Notice that the DAC driver uses mutex lock to ensure the thread safety, thus the APIs except `dac_oneshot_output_voltage()` are not allowed to be used in ISR.

**Kconfig Options**

- `CONFIG_DAC_ISR_IRAM_SAFE` controls whether the default ISR handler can work when cache is disabled. See [IRAM Safe](#) for more information.
- `CONFIG_DAC_SUPPRESS_DEPRECATE_WARN` controls whether to suppress the warning message compilation while using the legacy DAC driver.
- `CONFIG_DAC_ENABLE_DEBUG_LOG` is used to enable the debug log output. Enable this option increases the firmware binary size.

**Application Example**

The basic examples for the **One-shot Mode**, **Continuous Mode**, and **Cosine Mode** can be found in:

- `peripherals/dac/dac_oneshot`
- `peripherals/dac/dac_continuous`
- `peripherals/dac/dac_cosine_wave`

**API Reference**

**Header File**

- `components/driver/dac/include/driver/dac_oneshot.h`

This header file can be included with:

```c
#include "driver/dac_oneshot.h"
```

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```cmake
REQUIRES driver
```

or

```cmake
PRIV_REQUIRES driver
```

**Functions**

```c
esp_err_t dac_oneshot_new_channel (const dac_oneshot_config_t *oneshot_cfg, dac_oneshot_handle_t *ret_handle)
```

Allocate a new DAC oneshot channel.

**Note:** The channel will be enabled as well when the channel allocated

**Parameters**
**oneshot_cfg**  [in] The configuration for the oneshot channel

**ret_handle**  [out] The returned oneshot channel handle

**Returns**

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The DAC channel has been registered already
- ESP_ERR_NO_MEM No memory for the DAC oneshot channel resources
- ESP_OK Allocates the new DAC oneshot channel success

```c
esp_err_t dac_oneshot_del_channel(dac_oneshot_handle_t handle)
```

Delete the DAC oneshot channel.

**Note:** The channel will be disabled as well when the channel deleted

**Parameters**

- **handle**  [in] The DAC oneshot channel handle

**Returns**

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channel has already been de-registered
- ESP_OK Delete the oneshot channel success

```c
esp_err_t dac_oneshot_output_voltage(dac_oneshot_handle_t handle, uint8_t digi_value)
```

Output the voltage.

**Note:** Generally it’ll take 7~11 us on ESP32 and 10~21 us on ESP32-S2

**Parameters**

- **handle**  [in] The DAC oneshot channel handle
- **digi_value**  [in] The digital value that need to be converted

**Returns**

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_OK Convert the digital value success

**Structures**

```c
struct dac_oneshot_config_t
```

DA Coneshot channel configuration.

**Public Members**

```c
dac_channel_t chan_id
```

DAC channel id

**Type Definitions**

typedef struct dac_oneshot_s *dac_oneshot_handle_t

DAC oneshot channel handle

**Header File**

- components/driver/dac/include/driver/dac_cosine.h
- This header file can be included with:
#include "driver/dac_cosine.h"

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```cmake
REQUIRES driver
```

or

```cmake
PRIV_REQUIRES driver
```

## Functions

### esp_err_t dac_cosine_new_channel(const dac_cosine_config_t *cos_cfg, dac_cosine_handle_t *ret_handle)

Allocate a new DAC cosine wave channel.

**Note:** Since there is only one cosine wave generator, only the first channel can set the frequency of the cosine wave. Normally, the latter one is not allowed to set a different frequency, but the it can be forced to set by setting the bit `force_set_freq` in the configuration, notice that another channel will be affected as well when the frequency is updated.

**Parameters**
- `cos_cfg` - [in] The configuration of cosine wave channel
- `ret_handle` - [out] The returned cosine wave channel handle

**Returns**
- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The DAC channel has been registered already
- ESP_ERR_NO_MEM No memory for the DAC cosine wave channel resources
- ESP_OK Allocate the new DAC cosine wave channel success

### esp_err_t dac_cosine_del_channel(dac_cosine_handle_t handle)

Delete the DAC cosine wave channel.

**Parameters**
- `handle` - [in] The DAC cosine wave channel handle

**Returns**
- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channel has already been deregistered
- ESP_OK Delete the cosine wave channel success

### esp_err_t dac_cosine_start(dac_cosine_handle_t handle)

Start outputting the cosine wave on the channel.

**Parameters**
- `handle` - [in] The DAC cosine wave channel handle

**Returns**
- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channel has been started already
- ESP_OK Start the cosine wave success

### esp_err_t dac_cosine_stop(dac_cosine_handle_t handle)

Stop outputting the cosine wave on the channel.

**Parameters**
- `handle` - [in] The DAC cosine wave channel handle

**Returns**
- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channel has been stopped already
- ESP_OK Stop the cosine wave success
Structures

struct dac_cosine_config_t
DAC cosine channel configurations.

Public Members

dac_channel_t chan_id
The cosine wave channel id

uint32_t freq_hz
The frequency of cosine wave, unit: Hz. The cosine wave generator is driven by RTC_FAST clock which is divide from RC_FAST. With the default RTC clock, the minimum frequency of cosine wave is about 130 Hz. Although it can support up to several MHz frequency theoretically, the waveform will distort at high frequency due to the hardware limitation. Typically not suggest to set the frequency higher than 200 KHz

dac_cosine_clk_src_t clk_src
The clock source of the cosine wave generator, currently only support DAC_COSINE_CLK_SRC_DEFAULT

dac_cosine_atten_t atten
The attenuation of cosine wave amplitude

dac_cosine_phase_t phase
The phase of cosine wave, can only support DAC_COSINE_PHASE_0 or DAC_COSINE_PHASE_180, default as 0 while setting an unsupported phase

int8_t offset
The DC offset of cosine wave

bool force_set_freq
Force to set the cosine wave frequency

struct dac_cosine_config_t::[anonymous] flags
Flags of cosine mode

Type Definitions

typedef struct dac_cosine_s *dac_cosine_handle_t
DAC cosine wave channel handle

Header File

- components/driver/dac/include/driver/dac_continuous.h
- This header file can be included with:

```
#include "driver/dac_continuous.h"
```
- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
Chapter 2. API Reference

```c
REQUIRES driver
or
PRIV_REQUIRES driver
```

### Functions

**`esp_err_t dac_continuous_new_channels(const dac_continuous_config_t *cont_cfg, dac_continuous_handle_t *ret_handle)`**

Allocate new DAC channels in continuous mode.

**Note:** The DAC channels can’t be registered to continuous mode separately

#### Parameters

- `cont_cfg` - [in] Continuous mode configuration
- `ret_handle` - [out] The returned continuous mode handle

#### Returns

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The DAC channel has been registered already
- ESP_ERR_NOT_FOUND Not found the available dma peripheral, might be occupied
- ESP_ERR_NO_MEM No memory for the DAC continuous mode resources
- ESP_OK Allocate the new DAC continuous mode success

**`esp_err_t dac_continuous_del_channels(dac_continuous_handle_t handle)`**

Delete the DAC continuous handle.

**Parameters** `handle` - [in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`

#### Returns

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channels have already been deregistered or not disabled
- ESP_OK Delete the continuous channels success

**`esp_err_t dac_continuous_enable(dac_continuous_handle_t handle)`**

Enabled the DAC continuous mode.

**Note:** Must enable the channels before

#### Parameters

- `handle` - [in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`

#### Returns

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channels have been enabled already
- ESP_OK Enable the continuous output success

**`esp_err_t dac_continuous_disable(dac_continuous_handle_t handle)`**

Disable the DAC continuous mode.

#### Parameters

- `handle` - [in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`

#### Returns

- ESP_ERR_INVALID_ARG The input parameter is invalid
- ESP_ERR_INVALID_STATE The channels have been enabled already
- ESP_OK Disable the continuous output success
**esp_err_t** **dac_continuous_write** ([*dac_continuous_handle_t*](#) handle, uint8_t *buf, size_t buf_size, size_t *bytes_loaded, int timeout_ms)

Write DAC data continuously.

**Note:** The data in buffer will only be converted one time. This function will be blocked until all data loaded or timeout then the DAC output will keep outputting the voltage of the last data in the buffer.

**Note:** Specially, on ESP32, the data bit width of DAC continuous data is fixed to 16 bits while only the high 8 bits are available. The driver will help to expand the inputted buffer automatically by default, you can also align the data to 16 bits manually by clearing `CONFIG_DAC_DMA_AUTO_16BIT_ALIGN` in menuconfig.

**Parameters**
- **handle** —[in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`
- **buf** —[in] The digital data buffer to convert
- **buf_size** —[in] The buffer size of digital data buffer
- **bytes_loaded** —[out] The bytes that has been loaded into DMA buffer, can be NULL if don’t need it
- **timeout_ms** —[in] The timeout time in millisecond, set a minus value means will block forever

**Returns**
- **ESP_ERR_INVALID_ARG** The input parameter is invalid
- **ESP_ERR_INVALID_STATE** The DAC continuous mode has not been enabled yet
- **ESP_ERR_TIMEOUT** Waiting for semaphore or message queue timeout
- **ESP_OK** Success to output the acyclic DAC data

**esp_err_t** **dac_continuous_write_cyclically** ([*dac_continuous_handle_t*](#) handle, uint8_t *buf, size_t buf_size, size_t *bytes_loaded)

Write DAC continuous data cyclically.

**Note:** The data in buffer will be converted cyclically using DMA once this function is called, This function will return once the data loaded into DMA buffers.

**Note:** The buffer size of cyclically output is limited by the descriptor number and dma buffer size while initializing the continuous mode. Concretely, in order to load all the data into descriptors, the cyclic buffer size is not supposed to be greater than `desc_num * buf_size`

**Note:** Specially, on ESP32, the data bit width of DAC continuous data is fixed to 16 bits while only the high 8 bits are available. The driver will help to expand the inputted buffer automatically by default, you can also align the data to 16 bits manually by clearing `CONFIG_DAC_DMA_AUTO_16BIT_ALIGN` in menuconfig.

**Parameters**
- **handle** —[in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`
- **buf** —[in] The digital data buffer to convert
- **buf_size** —[in] The buffer size of digital data buffer
- **bytes_loaded** —[out] The bytes that has been loaded into DMA buffer, can be NULL if don’t need it

**Returns**
- **ESP_ERR_INVALID_ARG** The input parameter is invalid
• ESP_ERR_INVALID_STATE The DAC continuous mode has not been enabled yet
• ESP_OK Success to output the acyclic DAC data

`esp_err_t dac_continuous_register_event_callback (dac_continuous_handle_t handle, const dac_event_callbacks_t *callbacks, void *user_data)`

Set event callbacks for DAC continuous mode.

**Note:** User can deregister a previously registered callback by calling this function and setting the callback member in the callbacks structure to NULL.

**Note:** When CONFIG_DAC_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in this function, including the user_data, should be in the internal RAM as well.

**Parameters**
- `handle`  - [in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`
- `callbacks`  - [in] Group of callback functions, input NULL to clear the former callbacks
- `user_data`  - [in] User data, which will be passed to callback functions directly

**Returns**
- ESP_OK Set event callbacks successfully
- ESP_ERR_INVALID_ARG Set event callbacks failed because of invalid argument

`esp_err_t dac_continuous_start_async_writing (dac_continuous_handle_t handle)`

Start the async writing.

**Note:** When the asynchronous writing start, the DAC will keep outputting ‘0’ until the data are loaded into the DMA buffer. To loaded the data into DMA buffer, ‘on_convert_done’ callback is required, which can be registered by `dac_continuous_register_event_callback` before enabling

**Parameters**
- `handle`  - [in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`

**Returns**
- ESP_OK Start asynchronous writing successfully
- ESP_ERR_INVALID_ARG The handle is NULL
- ESP_ERR_INVALID_STATE The channel is not enabled or the ‘on_convert_done’ callback is not registered

`esp_err_t dac_continuous_stop_async_writing (dac_continuous_handle_t handle)`

Stop the sync writing.

**Parameters**
- `handle`  - [in] The DAC continuous channel handle that obtained from `dac_continuous_new_channels`

**Returns**
- ESP_OK Stop asynchronous writing successfully
- ESP_ERR_INVALID_ARG The handle is NULL
- ESP_ERR_INVALID_STATE Asynchronous writing has not started

`esp_err_t dac_continuous_write_asynchronously (dac_continuous_handle_t handle, uint8_t *dma_buf, size_t dma_buf_len, const uint8_t *data, size_t data_len, size_t *bytes_loaded)`

Write DAC data asynchronously.
Note: This function can be called when the asynchronous writing started, and it can be called in the callback directly but recommend to writing data in a task, referring to :example: :peripherals/dac/
dac_continuous/dac_audio

Parameters

- **handle** - [in] The DAC continuous channel handle that obtained from ‘dac_continuous_new_channels’
- **dma_buf** - [in] The DMA buffer address, it can be acquired from ‘dac_event_data_t’ in the ‘on_convert_done’ callback
- **dma_buf_len** - [in] The DMA buffer length, it can be acquired from ‘dac_event_data_t’ in the ‘on_convert_done’ callback
- **data** - [in] The data that need to be written
- **data_len** - [in] The data length the need to be written
- **bytes_loaded** - [out] The bytes number that has been loaded/written into the DMA buffer

Returns

- **ESP_OK** Write the data into DMA buffer successfully
- **ESP_ERR_INVALID_ARG** NULL pointer
- **ESP_ERR_INVALID_STATE** The channels haven’t start the asynchronous writing
- **ESP_ERR_NOT_FOUND** The param ‘dam_buf’ not match any existed DMA buffer

Structures

```c
struct dac_continuous_config_t
{
    DAC continuous channels’ configurations.
}
```

Public Members

```c

dac_channel_mask_t chan_mask
{
    DAC channels’ mask for selecting which channels are used
}
```

```c

uint32_t desc_num
{
    The number of DMA descriptor, at least 2 descriptors are required The number of descriptors is directly proportional to the max data buffer size while converting in cyclic output but only need to ensure it is greater than 1 in acyclic output Typically, suggest to set the number bigger than 5, in case the DMA stopped while sending a short buffer
}
```

```c

size_t buf_size
{
    The DMA buffer size, should be within 32–4092 bytes. Each DMA buffer will be attached to a DMA descriptor, i.e., the number of DMA buffer will be equal to the DMA descriptor number The DMA buffer size is not allowed to be greater than 4092 bytes The total DMA buffer size equal to desc_num * buf_size Typically, suggest to set the size to the multiple of 4
}
```

```c

uint32_t freq_hz
{
    The frequency of DAC conversion in continuous mode, unit: Hz The supported range is related to the target and the clock source. For the clock DAC_DIGI_CLK_SRC_DEFAULT: the range is 19.6 KHz to several MHz on ESP32 and 77 Hz to several MHz on ESP32-S2. For the clock DAC_DIGI_CLK_SRC_APLL: the range is 648 Hz to several MHz on ESP32 and 6 Hz to several MHz on ESP32-S2. Typically not suggest to set the frequency higher than 2 MHz, otherwise the severe distortion will appear
}
int8_t offset
The offset of the DAC digital data. Range -128~127

dac_continuous_digi_clk_src_t clk_src
The clock source of digital controller, which can affect the range of supported frequency. Currently
DAC_DIGI_CLK_SRC_DEFAULT and DAC_DIGI_CLK_SRC_APLL are available

dac_continuous_channel_mode_t chan_mode
The channel mode of continuous mode, only take effect when multiple channels enabled, depends con-
verting the buffer alternately or simultaneously

struct dac_event_data_t
Event structure used in DAC event queue.

Public Members

void *buf
The pointer of DMA buffer that just finished sending

size_t buf_size
The writable buffer size of the DMA buffer, equal to ‘dac_continuous_config_t::buf_size’

size_t write_bytes
The number of bytes that be written successfully

struct dac_event_callbacks_t
Group of DAC callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_DAC_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it
should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

dac_isr_callback_t on_convert_done
Callback of data conversion done event. An event data buffer previously loaded to the driver has been
output and converted. The event data includes DMA buffer address and size that just finished converting.

dac_isr_callback_t on_stop
Callback of finished sending all the data. All loaded event data buffers are converted. Driver is pending
for new data buffers to be loaded. The event data will be NULL in this callback.

Type Definitions
typedef struct dac_continuous_s *dac_continuous_handle_t
DAC continuous channel handle
typedef bool (*dac_isr_callback_t)(dac_continuous_handle_t handle, const dac_event_data_t *event, void *user_data)

DAC event callback.

Param handle [in] DAC channel handle, created from dac_continuous_new_channels()
Param event [in] DAC event data
Param user_data [in] User registered context, passed from dac_continuous_register_event_callback()

Return Whether a high priority task has been waken up by this callback function

Enumerations

enum dac_channel_mask_t

DAC channel mask.

Values:

enumerator DAC_CHANNEL_MASK_CH0
    DAC channel 0 is GPIO25(ESP32) / GPIO17(ESP32S2)

denumerator DAC_CHANNEL_MASK_CH1
    DAC channel 1 is GPIO26(ESP32) / GPIO18(ESP32S2)

denumerator DAC_CHANNEL_MASK_ALL
    Both DAC channel 0 and channel 1

Header File

- components/driver/dac/include/driver/dac_types.h
- This header file can be included with:

```c
#include "driver/dac_types.h"
```

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```cmake
REQUIRES driver
```

or

```cmake
PRIV_REQUIRES driver
```

Type Definitions

typedef soc_periph_dac_digi_clk_src_t dac_continuous_digi_clk_src_t

DAC DMA (digital controller) clock source.

typedef soc_periph_dac_cosine_clk_src_t dac_cosine_clk_src_t

DAC cosine wave generator clock source.

Enumerations

enum dac_continuous_channel_mode_t

DAC channel work mode in dma mode.
**Note:** Only take effect when multiple channels enabled.

**Note:** Assume the data in buffer is ‘A B C D E F’

- DAC_CHANNEL_MODE_SIMUL:
  - channel 0: A B C D E F
  - channel 1: A B C D E F

- DAC_CHANNEL_MODE_ALTER:
  - channel 0: A C E
  - channel 1: B D F

**Values:**

**enumerator** DAC_CHANNEL_MODE_SIMUL

The data in the DMA buffer is simultaneously output to the enable channel of the DAC.

**enumerator** DAC_CHANNEL_MODE_ALTER

The data in the DMA buffer is alternately output to the enable channel of the DAC.

**Header File**

- components/hal/include/hal/dac_types.h
- This header file can be included with:

```
#include "hal/dac_types.h"
```

**Enumerations**

**enum** dac_channel_t

**Values:**

**enumerator** DAC_CHAN_0

DAC channel 0 is GPIO25(ESP32) / GPIO17(ESP32S2)

**enumerator** DAC_CHAN_1

DAC channel 1 is GPIO26(ESP32) / GPIO18(ESP32S2)

**enumerator** DAC_CHANNEL_1

Alias of ‘DAC_CHAN_0’, now the channel index start from ‘0’

**enumerator** DAC_CHANNEL_2

Alias of ‘DAC_CHAN_1’, now the channel index start from ‘0’

**enum** dac_cosine_atten_t

The attenuation of the amplitude of the cosine wave generator. The max amplitude is VDD3P3_RTC.

**Values:**

**enumerator** DAC_COSINE_ATTEN_DEFAULT

No attenuation to the DAC cosine wave amplitude. Default.

**enumerator** DAC_COSINE_ATTEN_DB_0

Original amplitude of the DAC cosine wave, equals to DAC_COSINE_ATTEN_DEFAULT
enumerator DAC_COSINE_ATTEN_DB_6
1/2 amplitude of the DAC cosine wave

enumerator DAC_COSINE_ATTEN_DB_12
1/4 amplitude of the DAC cosine wave

enumerator DAC_COSINE_ATTEN_DB_18
1/8 amplitude of the DAC cosine wave

enum dac_cosine_phase_t
Set the phase of the cosine wave generator output.

Note: Only 0 or 180 are supported, it will be set to 0 as default if configured to an unsupported phase.

Values:

everenumerator DAC_COSINE_PHASE_0
Phase shift +0°

enumerator DAC_COSINE_PHASE_180
Phase shift +180°

2.5.6 GPIO & RTC GPIO

GPIO Summary

The ESP32-S2 chip features 43 physical GPIO pins (GPIO0 ~ GPIO21 and GPIO26 ~ GPIO46). Each pin can be used as a general-purpose I/O, or be connected to an internal peripheral signal. Through IO MUX, RTC IO MUX and the GPIO matrix, peripheral input signals can be from any IO pins, and peripheral output signals can be routed to any IO pins. Together these modules provide highly configurable I/O. For more details, see ESP32-S2 Technical Reference Manual > IO MUX and GPIO Matrix (GPIO, IO_MUX) [PDF].

The table below provides more information on pin usage, and please note the comments in the table for GPIOs with restrictions.

<table>
<thead>
<tr>
<th>GPIO</th>
<th>Analog Function</th>
<th>RTC GPIO</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO0</td>
<td></td>
<td>RTC_GPIO0</td>
<td>Strapping pin</td>
</tr>
<tr>
<td>GPIO1</td>
<td>ADC1_CH0</td>
<td>RTC_GPIO1</td>
<td></td>
</tr>
<tr>
<td>GPIO2</td>
<td>ADC1_CH1</td>
<td>RTC_GPIO2</td>
<td></td>
</tr>
<tr>
<td>GPIO3</td>
<td>ADC1_CH2</td>
<td>RTC_GPIO3</td>
<td></td>
</tr>
<tr>
<td>GPIO4</td>
<td>ADC1_CH3</td>
<td>RTC_GPIO4</td>
<td></td>
</tr>
<tr>
<td>GPIO5</td>
<td>ADC1_CH4</td>
<td>RTC_GPIO5</td>
<td></td>
</tr>
<tr>
<td>GPIO6</td>
<td>ADC1_CH5</td>
<td>RTC_GPIO6</td>
<td></td>
</tr>
<tr>
<td>GPIO7</td>
<td>ADC1_CH6</td>
<td>RTC_GPIO7</td>
<td></td>
</tr>
<tr>
<td>GPIO8</td>
<td>ADC1_CH7</td>
<td>RTC_GPIO8</td>
<td></td>
</tr>
<tr>
<td>GPIO9</td>
<td>ADC1_CH8</td>
<td>RTC_GPIO9</td>
<td></td>
</tr>
<tr>
<td>GPIO10</td>
<td>ADC1_CH9</td>
<td>RTC_GPIO10</td>
<td></td>
</tr>
<tr>
<td>GPIO11</td>
<td>ADC2_CH0</td>
<td>RTC_GPIO11</td>
<td></td>
</tr>
<tr>
<td>GPIO12</td>
<td>ADC2_CH1</td>
<td>RTC_GPIO12</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
### Table 2 - continued from previous page

<table>
<thead>
<tr>
<th>GPIO</th>
<th>Analog Function</th>
<th>RTC_GPIO</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO13</td>
<td>ADC2_CH2</td>
<td>RTC_GPIO13</td>
<td></td>
</tr>
<tr>
<td>GPIO14</td>
<td>ADC2_CH3</td>
<td>RTC_GPIO14</td>
<td></td>
</tr>
<tr>
<td>GPIO15</td>
<td>ADC2_CH4</td>
<td>RTC_GPIO15</td>
<td></td>
</tr>
<tr>
<td>GPIO16</td>
<td>ADC2_CH5</td>
<td>RTC_GPIO16</td>
<td></td>
</tr>
<tr>
<td>GPIO17</td>
<td>ADC2_CH6</td>
<td>RTC_GPIO17</td>
<td></td>
</tr>
<tr>
<td>GPIO18</td>
<td>ADC2_CH7</td>
<td>RTC_GPIO18</td>
<td></td>
</tr>
<tr>
<td>GPIO19</td>
<td>ADC2_CH8</td>
<td>RTC_GPIO19</td>
<td></td>
</tr>
<tr>
<td>GPIO20</td>
<td>ADC2_CH9</td>
<td>RTC_GPIO20</td>
<td></td>
</tr>
<tr>
<td>GPIO21</td>
<td></td>
<td>RTC_GPIO21</td>
<td></td>
</tr>
<tr>
<td>GPIO26</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO27</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO28</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO29</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO30</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO31</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO32</td>
<td></td>
<td>SPI0/1</td>
<td></td>
</tr>
<tr>
<td>GPIO33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO39</td>
<td></td>
<td>JTAG</td>
<td></td>
</tr>
<tr>
<td>GPIO40</td>
<td></td>
<td>JTAG</td>
<td></td>
</tr>
<tr>
<td>GPIO41</td>
<td></td>
<td>JTAG</td>
<td></td>
</tr>
<tr>
<td>GPIO42</td>
<td></td>
<td>JTAG</td>
<td></td>
</tr>
<tr>
<td>GPIO43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO45</td>
<td></td>
<td>Strapping pin</td>
<td></td>
</tr>
<tr>
<td>GPIO46</td>
<td></td>
<td>GPI: Strapping pin</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
- Strapping pin: GPIO0, GPIO45 and GPIO46 are strapping pins. For more information, please refer to [ESP32-S2 datasheet](#).
- SPI0/1: GPIO26-32 are usually used for SPI flash and PSRAM and not recommended for other uses.
- JTAG: GPIO39-42 are usually used for inline debug.
- GPI: GPIO46 is fixed to pull-down and is input only.

GPIO driver offers a dump function `gpio_dump_io_configuration()` to show the configurations of the IOs at the moment, such as pull-up / pull-down, input / output enable, pin mapping etc. Below is an example dump:

```
=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=✻
Chapter 2. API Reference

(continued from previous page)

<table>
<thead>
<tr>
<th>GPIO Matrix SigOut ID: 256 {simple GPIO output}</th>
</tr>
</thead>
<tbody>
<tr>
<td>SleepSelEn: 1</td>
</tr>
<tr>
<td>IO[26] <strong>RESERVED</strong> -</td>
</tr>
<tr>
<td>Pullup: 1, Pulldown: 0, DriveCap: 2</td>
</tr>
<tr>
<td>InputEn: 1, OutputEn: 0, OpenDrain: 0</td>
</tr>
<tr>
<td>FuncSel: 0 (IOMUX)</td>
</tr>
<tr>
<td>SleepSelEn: 1</td>
</tr>
</tbody>
</table>

------------------------- IO DUMP End -------------------------

If an IO pin is routed to a peripheral signal through the GPIO matrix, the signal ID printed in the dump information is defined in the soc/gpio_sig_map.h file. The word **RESERVED** indicates the IO is occupied by either FLASH or PSRAM. It is strongly not recommended to reconfigure them for other application purposes.

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 mode
- The Ultra Low Power co-processor is running
- Analog functions such as ADC/DAC/etc are in use

GPIO Glitch Filter

The ESP32-S2 chip features hardware filters to remove unwanted glitch pulses from the input GPIO, which can help reduce false triggering of the interrupt and prevent a noise being routed to the peripheral side.

Each GPIO can be configured with a glitch filter, which can be used to filter out pulses shorter than two sample clock cycles. The duration of the filter is not configurable. The sample clock is the clock source of the IO_MUX. In the driver, we call this kind of filter as pin glitch filter. You can create the filter handle by calling `gpio_new_pin_glitch_filter()`. All the configurations for a pin glitch filter are listed in the `gpio_pin_glitch_filter_config_t` structure.

- `gpio_pin_glitch_filter_config_t::gpio_num` sets the GPIO number to enable the glitch filter.

The glitch filter is disabled by default, and can be enabled by calling `gpio_glitch_filter_enable()`. To recycle the filter, you can call `gpio_del_glitch_filter()`. Please note, before deleting the filter, you should disable it first by calling `gpio_glitch_filter_disable()`.

Application Example

- GPIO output and input interrupt example: peripherals/gpio/generic_gpio.

API Reference - Normal GPIO

Header File

- `components/esp_driver_gpio/include/driver/gpio.h`
- This header file can be included with:

```
#include "driver/gpio.h"
```

- This header file is a part of the API provided by the `esp_driver_gpio` component. To declare that your component depends on `esp_driver_gpio`, add the following to your CMakeLists.txt:
REQUIRES esp_driver_gpio
or

PRIV_REQUIRES esp_driver_gpio

Functions

```c
esp_err_t gpio_config (const gpio_config_t *pGPIOConfig)
```

GPIO common configuration.

- **Parameters**
  - `pGPIOConfig` – Pointer to GPIO configure struct

- **Returns**
  - ESP_OK success
  - ESP_ERR_INVALID_ARG Parameter error

```c
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).

- **Parameters**
  - `gpio_num` – GPIO number.

- **Returns**
  - Always return ESP_OK.

```c
esp_err_t gpio_set_intr_type (gpio_num_t gpio_num, gpio_int_type_t intr_type)
```

GPIO set interrupt trigger type.

- **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

- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG Parameter error

```c
esp_err_t gpio_intr_enable (gpio_num_t gpio_num)
```

Enable GPIO module interrupt signal.

- **Parameters**
  - `gpio_num` – GPIO number. If you want to enable an interrupt on e.g. GPIO16, `gpio_num` should be GPIO_NUM_16 (16);

- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG Parameter error

---

**Note:**
- This function also configures the IOMUX for this pin to the GPIO function, and disconnects any other peripheral output configured via GPIO Matrix.

**Note:**
- ESP32: Please do not use the interrupt of GPIO36 and GPIO39 when using ADC or Wi-Fi and Bluetooth with sleep mode enabled. Please refer to the comments of `adc1_get_raw`. Please refer to Section 3.11 of *ESP32 ECO and Workarounds for Bugs* for the description of this issue.

---
**esp_err_t gpio_intr_disable** (gpio_num_t gpio_num)

Disable GPIO module interrupt signal.

**Note:** This function is allowed to be executed when Cache is disabled within ISR context, by enabling **CONFIG_GPIO_CTRL_FUNC_IN_IRAM**

**Parameters**
- **gpio_num** – GPIO number. If you want to disable the interrupt of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);

**Returns**
- ESP_OK success
- ESP_ERR_INVALID_ARG Parameter error

**esp_err_t gpio_set_level** (gpio_num_t gpio_num, uint32_t level)

GPIO set output level.

**Note:** This function is allowed to be executed when Cache is disabled within ISR context, by enabling **CONFIG_GPIO_CTRL_FUNC_IN_IRAM**

**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

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO number error

**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.

**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);

**Returns**
- 0 the GPIO input level is 0
- 1 the GPIO input level is 1

**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

**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

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO error

**esp_err_t gpio_set_pull_mode** (gpio_num_t gpio_num, gpio_pull_mode_t pull)

Configure GPIO pull-up/pull-down resistors.
Note: ESP32: Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

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.

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG : Parameter error

```c
esp_err_t gpio_wakeup_enable (gpio_num_t gpio_num, gpio_int_type_t intr_type)
```
Enable GPIO wake-up function.

Parameters
• `gpio_num` – GPIO number.
• `intr_type` – GPIO wake-up type. Only `GPIO_INTR_LOW_LEVEL` or `GPIO_INTR_HIGH_LEVEL` can be used.

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

```c
esp_err_t gpio_wakeup_disable (gpio_num_t gpio_num)
```
Disable GPIO wake-up function.

Parameters `gpio_num` – GPIO number

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

```c
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.
• `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.

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG GPIO error
• ESP_ERR_NOT_FOUND No free interrupt found with the specified flags

```c
esp_err_t gpio_pullup_en (gpio_num_t gpio_num)
```
Enable pull-up on GPIO.

Parameters `gpio_num` – GPIO number

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error
**esp_err_t gpio_pullup_dis**(gpio_num_t gpio_num)
Disable pull-up on GPIO.

**Parameters**
gpio_num – GPIO number

**Returns**
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

**esp_err_t gpio_pulldown_en**(gpio_num_t gpio_num)
Enable pull-down on GPIO.

**Parameters**
gpio_num – GPIO number

**Returns**
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

**esp_err_t gpio_pulldown_dis**(gpio_num_t gpio_num)
Disable pull-down on GPIO.

**Parameters**
gpio_num – GPIO number

**Returns**
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

**esp_err_t gpio_install_isr_service**(int intr_alloc_flags)
Install the GPIO driver’s ETS_GPIO_INTR_SOURCE 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.

**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.

**Returns**
• ESP_OK Success
• ESP_ERR_NO_MEM Nomemory 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

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.

**Parameters**
• gpio_num – GPIO number
• isr_handler – ISR handler function for the corresponding GPIO number.
• args – parameter for ISR handler.

**Returns**
• ESP_OK Success
• ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
• ESP_ERR_INVALID_ARG Parameter error
Chapter 2. API Reference

```
esp_err_t gpio_isr_handler_remove (gpio_num_t gpio_num)

Remove ISR handler for the corresponding GPIO pin.

Parameters gpio_num – GPIO number

Returns

• ESP_OK Success
• ESP_ERR_INVALID_STATE Wrong state, the ISR service has not been initialized.
• ESP_ERR_INVALID_ARG Parameter error
```

```
esp_err_t gpio_set_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t strength)

Set GPIO pad drive capability.

Parameters

• gpio_num – GPIO number, only support output GPIOs
• strength – Drive capability of the pad

Returns

• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error
```

```
esp_err_t gpio_get_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t *strength)

Get GPIO pad drive capability.

Parameters

• gpio_num – GPIO number, only support output GPIOs
• strength – Pointer to accept drive capability of the pad

Returns

• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error
```

```
esp_err_t gpio_hold_en (gpio_num_t gpio_num)

Enable gpio pad hold function.

When a GPIO is set to hold, its state is latched at that moment and will not change when the internal signal or
the IO MUX/GPIO configuration is modified (including input enable, output enable, output value, function,
and drive strength values). This function can be used to retain the state of GPIOs when the chip or system is
reset, for example, when watchdog time-out or Deep-sleep events are triggered.

This function works in both input and output modes, and only applicable to output-capable GPIOs. If this
function is enabled: in output mode: the output level of the GPIO will be locked and can not be changed. in
input mode: the input read value can still reflect the changes of the input signal.

However, on ESP32/S2/C3/S3/C2, this function cannot be used to hold the state of a digital GPIO during
Deep-sleep. Even if this function is enabled, the digital GPIO will be reset to its default state when the chip
wakes up from Deep-sleep. If you want to hold the state of a digital GPIO during Deep-sleep, please call
gpio_deep_sleep_hold_en.

Power down or call gpio_hold_dis will disable this function.

Parameters gpio_num – GPIO number, only support output-capable GPIOs

Returns

• ESP_OK Success
• ESP_ERR_NOT_SUPPORTED Not support pad hold function
```

```
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 hight level
before calling gpio_hold_dis.

Parameters gpio_num – GPIO number, only support output-capable GPIOs
```
Chapter 2. API Reference

Returns

• ESP_OK Success
• ESP_ERR_NOT_SUPPORTED Not support pad hold function

void gpio_deep_sleep_hold_en (void)

Enable all digital gpio pads hold function during Deep-sleep.

Enabling this feature makes all digital gpio pads be at the holding state during Deep-sleep. The state of each pad holds is its active configuration (not pad’s sleep configuration!).

Note that this pad hold feature only works when the chip is in Deep-sleep mode. When the chip is in active mode, the digital gpio state can be changed freely even you have called this function.

After this API is being called, the digital gpio Deep-sleep hold feature will work during every sleep process. You should call gpio_deep_sleep_hold_dis to disable this feature.

void gpio_deep_sleep_hold_dis (void)

Disable all digital gpio pads hold function during Deep-sleep.

void gpio_iomux_in (uint32_t gpio_num, uint32_t signal_idx)

SOC_GPIO_SUPPORT_HOLD_SINGLE_IO_IN_DSLP.

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.

esp_err_t gpio_force_hold_all (void)

Force hold all digital and rtc gpio pads.

GPIO force hold, no matter the chip in active mode or sleep modes.

This function will immediately cause all pads to latch the current values of input enable, output enable, output value, function, and drive strength values.

Warning: This function will hold flash and UART pins as well. Therefore, this function, and all code run afterwards (till calling gpio_force_unhold_all to disable this feature), MUST be placed in internal RAM as holding the flash pins will halt SPI flash operation, and holding the UART pins will halt any UART logging.

esp_err_t gpio_force_unhold_all (void)

Force unhold all digital and rtc gpio pads.

gpio_sleep_sel_en (gpio_num_t gpio_num)

Enable SLP_SEL to change GPIO status automatically in lightsleep.

Parameters gpio_num – GPIO number of the pad.

Returns

• ESP_OK Success
Chapter 2. API Reference

```c
esp_err_t gpio_sleep_sel_dis (gpio_num_t gpio_num)

Disable SLP_SEL to change GPIO status automatically in lightsleep.

Parameters
  gpio_num - GPIO number of the pad.

Returns
  • ESP_OK Success
```

```c
esp_err_t gpio_sleep_set_direction (gpio_num_t gpio_num, gpio_mode_t mode)

GPIO set direction at sleep.

Configure GPIO direction, such as output_only, input_only, output_and_input

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

Returns
  • ESP_OK Success
  • ESP_ERR_INVALID_ARG GPIO error
```

```c
esp_err_t gpio_sleep_set_pull_mode (gpio_num_t gpio_num, gpio_pull_mode_t pull)

Configure GPIO pull-up/pull-down resistors at sleep.

Note: ESP32: Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

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 pullup/down mode.

Returns
  • ESP_OK Success
  • ESP_ERR_INVALID_ARG : Parameter error
```

```c
esp_err_t gpio_dump_io_configuration (FILE *out_stream, uint64_t io_bit_mask)

Dump IO configuration information to console.

Parameters
  • out_stream - IO stream (e.g. stdout)
  • io_bit_mask - IO pin bit mask, each bit maps to an IO

Returns
  • ESP_OK Success
  • ESP_ERR_INVALID_ARG Parameter error
```

### Structures

```c
struct gpio_config_t

Configuration parameters of GPIO pad for gpio_config function.
```

### Public Members

```c
uint64_t pin_bit_mask

GPIO pin: set with bit mask, each bit maps to a GPIO
```
Chapter 2. API Reference

```c
#include "hal/gpio_types.h"
```

```c
const gpio_mode_t mode

GPIO mode: set input/output mode
```

```c
const gpio_pullup_t pull_up_en

GPIO pull-up
```

```c
const gpio_pulldown_t pull_down_en

GPIO pull-down
```

```c
const gpio_int_type_t intr_type

GPIO interrupt type
```

Macros

```c
#define GPIO_PIN_COUNT

Check whether it is a valid GPIO number.
```

```c
#define GPIO_IS_VALID_GPIO(gpio_num)

Check whether it can be a valid GPIO number of output mode.
```

```c
#define GPIO_IS_VALID_DIGITAL_IO_PAD(gpio_num)

Check whether it can be a valid digital I/O pad.
```

Type Definitions

```c
typedef intr_handle_t gpio_isr_handle_t

typedef void (*gpio_isr_t)(void *arg)

GPIO interrupt handler.
```

```
Param arg User registered data
```

Header File

- components/hal/include/hal/gpio_types.h
- This header file can be included with:

```
#include "hal/gpio_types.h"
```

Macros

```
#define GPIO_PIN_REG_0
```

```
#define GPIO_PIN_REG_1
```

```
#define GPIO_PIN_REG_2
```

```
#define GPIO_PIN_REG_3
```

```
#define GPIO_PIN_REG_4
```
GPIO_PIN_REG_5
GPIO_PIN_REG_6
GPIO_PIN_REG_7
GPIO_PIN_REG_8
GPIO_PIN_REG_9
GPIO_PIN_REG_10
GPIO_PIN_REG_11
GPIO_PIN_REG_12
GPIO_PIN_REG_13
GPIO_PIN_REG_14
GPIO_PIN_REG_15
GPIO_PIN_REG_16
GPIO_PIN_REG_17
GPIO_PIN_REG_18
GPIO_PIN_REG_19
GPIO_PIN_REG_20
GPIO_PIN_REG_21
GPIO_PIN_REG_22
GPIO_PIN_REG_23
GPIO_PIN_REG_24
GPIO_PIN_REG_25
GPIO_PIN_REG_26
GPIO_PIN_REG_27
Chapter 2. API Reference

GPIO_PIN_REG_28
GPIO_PIN_REG_29
GPIO_PIN_REG_30
GPIO_PIN_REG_31
GPIO_PIN_REG_32
GPIO_PIN_REG_33
GPIO_PIN_REG_34
GPIO_PIN_REG_35
GPIO_PIN_REG_36
GPIO_PIN_REG_37
GPIO_PIN_REG_38
GPIO_PIN_REG_39
GPIO_PIN_REG_40
GPIO_PIN_REG_41
GPIO_PIN_REG_42
GPIO_PIN_REG_43
GPIO_PIN_REG_44
GPIO_PIN_REG_45
GPIO_PIN_REG_46
GPIO_PIN_REG_47
GPIO_PIN_REG_48
GPIO_PIN_REG_49
GPIO_PIN_REG_50
Chapter 2. API Reference

GPIO_PIN_REG_51
GPIO_PIN_REG_52
GPIO_PIN_REG_53
GPIO_PIN_REG_54
GPIO_PIN_REG_55
GPIO_PIN_REG_56

Enumerations

enum gpio_port_t

Values:

enumerator GPIO_PORT_0

enumerator GPIO_PORT_MAX

enum gpio_int_type_t

Values:

enumerator GPIO_INTR_DISABLE
    Disable GPIO interrupt

enumerator GPIO_INTR_POSEDGE
    GPIO interrupt type: rising edge

enumerator GPIO_INTR_NEGEDGE
    GPIO interrupt type: falling edge

enumerator GPIO_INTR_ANYEDGE
    GPIO interrupt type: both rising and falling edge

enumerator GPIO_INTR_LOW_LEVEL
    GPIO interrupt type: input low level trigger

enumerator GPIO_INTR_HIGH_LEVEL
    GPIO interrupt type: input high level trigger

enumerator GPIO_INTR_MAX

enum gpio_mode_t

Values:
enumerator **GPIO_MODE_DISABLE**
  
  GPIO mode : disable input and output

enumerator **GPIO_MODE_INPUT**
  
  GPIO mode : input only

enumerator **GPIO_MODE_OUTPUT**
  
  GPIO mode : output only mode

enumerator **GPIO_MODE_OUTPUT_OD**
  
  GPIO mode : output only with open-drain mode

enumerator **GPIO_MODE_INPUT_OUTPUT_OD**
  
  GPIO mode : output and input with open-drain mode

enumerator **GPIO_MODE_INPUT_OUTPUT**
  
  GPIO mode : output and input mode

enum **gpio_pullup_t**

  Values:

  enumerator **GPIO_PULLUP_DISABLE**
    
    Disable GPIO pull-up resistor

  enumerator **GPIO_PULLUP_ENABLE**
    
    Enable GPIO pull-up resistor

enum **gpio_pulldown_t**

  Values:

  enumerator **GPIO_PULLDOWN_DISABLE**
    
    Disable GPIO pull-down resistor

  enumerator **GPIO_PULLDOWN_ENABLE**
    
    Enable GPIO pull-down resistor

enum **gpio_pull_mode_t**

  Values:

  enumerator **GPIO_PULLUP_ONLY**
    
    Pad pull up

  enumerator **GPIO_PULLDOWN_ONLY**
    
    Pad pull down

  enumerator **GPIO_PULLUP_PULLDOWN**
    
    Pad pull up + pull down
enumerator GPIO_FLOATING
    Pad floating

enum gpio_drive_cap_t
    Values:
    
    enumerator GPIO_DRIVE_CAP_0
        Pad drive capability: weak
    
    enumerator GPIO_DRIVE_CAP_1
        Pad drive capability: stronger
    
    enumerator GPIO_DRIVE_CAP_2
        Pad drive capability: medium
    
    enumerator GPIO_DRIVE_CAP_DEFAULT
        Pad drive capability: medium
    
    enumerator GPIO_DRIVE_CAP_3
        Pad drive capability: strongest
    
    enumerator GPIO_DRIVE_CAP_MAX

API Reference - RTC GPIO

Header File

- components/esp_driver_gpio/include/driver/rtc_io.h
- This header file can be included with:

```c
#include "driver/rtc_io.h"
```

- This header file is a part of the API provided by the esp_driver_gpio component. To declare that your component depends on esp_driver_gpio, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_gpio

or

PRIV_REQUIRES esp_driver_gpio
```

Functions

bool rtc_gpio_is_valid_gpio (gpio_num_t gpio_num)
    Determine if the specified GPIO is a valid RTC GPIO.

    Parameters gpio_num - GPIO number
    Returns true if GPIO is valid for RTC GPIO use. false otherwise.

int rtc_io_number_get (gpio_num_t gpio_num)
    Get RTC IO index number by gpio number.

    Parameters gpio_num - GPIO number
    Returns >=0: Index of rtcio. -1 : The gpio is not rtcio.
**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.

**Parameters**
- **gpio_num** – GPIO number (e.g. GPIO_NUM_12)

**Returns**
- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**esp_err_t rtc_gpio_deinit (gpio_num_t gpio_num)**

Init a GPIO as digital GPIO.

**Parameters**
- **gpio_num** – GPIO number (e.g. GPIO_NUM_12)

**Returns**
- ESP_OK success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**uint32_t rtc_gpio_get_level (gpio_num_t gpio_num)**

Get the RTC IO input level.

**Parameters**
- **gpio_num** – GPIO number (e.g. GPIO_NUM_12)

**Returns**
- 1 High level
- 0 Low level
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**esp_err_t rtc_gpio_set_level (gpio_num_t gpio_num, uint32_t level)**

Set the RTC IO output level.

**Parameters**
- **gpio_num** – GPIO number (e.g. GPIO_NUM_12)
- **level** – output level

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**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.

**Parameters**
- **gpio_num** – GPIO number (e.g. GPIO_NUM_12)
- **mode** – GPIO direction

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**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 supports INPUT_ONLY mode. The rest targets support INPUT_ONLY, OUTPUT_ONLY, INPUT_OUTPUT mode.

**Parameters**
- **gpio_num** – GPIO number (e.g. GPIO_NUM_12)
- **mode** – GPIO direction

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO
**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.

- **Parameters**: gpio_num – GPIO number (e.g. GPIO_NUM_12)
- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**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.

- **Parameters**: gpio_num – GPIO number (e.g. GPIO_NUM_12)
- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**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.

- **Parameters**: gpio_num – GPIO number (e.g. GPIO_NUM_12)
- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**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.

- **Parameters**: gpio_num – GPIO number (e.g. GPIO_NUM_12)
- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**esp_err_t rtc_gpio_set_drive_capability** (gpio_num_t gpio_num, gpio_drive_cap_t strength)

Set RTC GPIO pad drive capability.

- **Parameters**
  - gpio_num – GPIO number, only support output GPIOs
  - strength – Drive capability of the pad
- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG Parameter error

**esp_err_t rtc_gpio_get_drive_capability** (gpio_num_t gpio_num, gpio_drive_cap_t *strength)

Get RTC GPIO pad drive capability.

- **Parameters**
  - gpio_num – GPIO number, only support output GPIOs
  - strength – Pointer to accept drive capability of the pad
- **Returns**
  - ESP_OK Success
  - ESP_ERR_INVALID_ARG Parameter error
**esp_err_t** `rtc_gpio_iomux_func_sel` (gpio_num_t gpio_num, int func)

Select a RTC IOMUX function for the RTC IO.

**Parameters**
- `gpio_num` - GPIO number
- `func` - Function to assign to the pin

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

**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.

**Parameters**
- `gpio_num` - GPIO number (e.g. GPIO_NUM_12)

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**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.

**Parameters**
- `gpio_num` - GPIO number (e.g. GPIO_NUM_12)

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO is not an RTC IO

**esp_err_t** `rtc_gpio_force_hold_en_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_wakeup_enable` (gpio_num_t gpio_num, gpio_int_type_t intr_type)

Enable wakeup from sleep mode using specific GPIO.

**Parameters**
- `gpio_num` - GPIO number
- `intr_type` - Wakeup on high level (GPIO_INTR_HIGH_LEVEL) or low level (GPIO_INTR_LOW_LEVEL)

**Returns**
- 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.

**esp_err_t** `rtc_gpio_wakeup_disable` (gpio_num_t gpio_num)

Disable wakeup from sleep mode using specific GPIO.

**Parameters**
- `gpio_num` - GPIO number

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if gpio_num is not an RTC IO
Chapter 2. API Reference

Macros

\texttt{RTC_GPIO_IS_VALID_GPIO} (gpio_num)

Header File

- \texttt{components/esp_driver_gpio/include/driver/lp_io.h}
- This header file can be included with:

\begin{verbatim}
#include "driver/lp_io.h"
\end{verbatim}

- This header file is a part of the API provided by the \texttt{esp_driver_gpio} component. To declare that your component depends on \texttt{esp_driver_gpio}, add the following to your \texttt{CMakeLists.txt}:

\begin{verbatim}
REQUIRES esp_driver_gpio
\end{verbatim}
or

\begin{verbatim}
PRIV_REQUIRES esp_driver_gpio
\end{verbatim}

Header File

- \texttt{components/hal/include/hal/rtc_io_types.h}
- This header file can be included with:

\begin{verbatim}
#include "hal/rtc_io_types.h"
\end{verbatim}

Enumerations

\texttt{enum rtc_gpio_mode_t}

\begin{verbatim}
RTCIO output/input mode type.
\end{verbatim}

\texttt{Values:}

- \texttt{enumerator RTC_GPIO_MODE_INPUTONLY}
  Pad input

- \texttt{enumerator RTC_GPIO_MODE_OUTPUTONLY}
  Pad output

- \texttt{enumerator RTC_GPIO_MODE_INPUTOUTPUT}
  Pad input + output

- \texttt{enumerator RTC_GPIO_MODE_DISABLED}
  Pad (output + input) disable

- \texttt{enumerator RTC_GPIO_MODE_OUTPUTOD}
  Pad open-drain output

- \texttt{enumerator RTC_GPIO_MODE_INPUTOUTPUTOD}
  Pad input + open-drain output
API Reference - GPIO Glitch Filter

Header File

- components/esp_driver_gpio/include/driver/gpio_filter.h
- This header file can be included with:

```c
#include "driver/gpio_filter.h"
```

- This header file is a part of the API provided by the esp_driver_gpio component. To declare that your component depends on esp_driver_gpio, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_gpio
```

or

```cmake
PRIV_REQUIRES esp_driver_gpio
```

Functions

`esp_err_t gpio_new_pin_glitch_filter` (const `gpio_pin_glitch_filter_config_t` *config, 
`gpio_glitch_filter_handle_t` *ret_filter)

Create a pin glitch filter.

**Note:** Pin glitch filter parameters are fixed, pulses shorter than two sample clocks (IO-MUX’s source clock) will be filtered out. It’s independent with “flex” glitch filter. See also `gpio_new_flex_glitch_filter`.

**Note:** The created filter handle can later be deleted by `gpio_del_glitch_filter`.

**Parameters**
- `config` [in] Glitch filter configuration
- `ret_filter` [out] Returned glitch filter handle

**Returns**
- ESP_OK: Create a pin glitch filter successfully
- ESP_ERR_INVALID_ARG: Create a pin glitch filter failed because of invalid arguments (e.g. wrong GPIO number)
- ESP_ERR_NO_MEM: Create a pin glitch filter failed because of out of memory
- ESP_FAIL: Create a pin glitch filter failed because of other error

`esp_err_t gpio_new_flex_glitch_filter` (const `gpio_flex_glitch_filter_config_t` *config, 
`gpio_glitch_filter_handle_t` *ret_filter)

Allocate a flex glitch filter.

**Note:** “flex” means the filter parameters (window, threshold) are adjustable. It’s independent with pin glitch filter. See also `gpio_new_pin_glitch_filter`.

**Note:** The created filter handle can later be deleted by `gpio_del_glitch_filter`.

**Parameters**
- `config` [in] Glitch filter configuration
- `ret_filter` [out] Returned glitch filter handle

**Returns**
- ESP_OK: Allocate a flex glitch filter successfully
• ESP_ERR_INVALID_ARG: Allocate a flex glitch filter failed because of invalid arguments (e.g. wrong GPIO number, filter parameters out of range)
• ESP_ERR_NO_MEM: Allocate a flex glitch filter failed because of out of memory
• ESP_ERR_NOT_FOUND: Allocate a flex glitch filter failed because the underlying hardware resources are used up
• ESP_FAIL: Allocate a flex glitch filter failed because of other error

**esp_err_t gpio_del_glitch_filter (gpio_glitch_filter_handle_t filter)**
Delete a glitch filter.

Parameters

- **filter** – [in] Glitch filter handle returned from `gpio_new_flex_glitch_filter` or `gpio_new_pin_glitch_filter`

Returns

- ESP_OK: Delete glitch filter successfully
- ESP_ERR_INVALID_ARG: Delete glitch filter failed because of invalid arguments
- ESP_ERR_INVALID_STATE: Delete glitch filter failed because the glitch filter is still in working
- ESP_FAIL: Delete glitch filter failed because of other error

**esp_err_t gpio_glitch_filter_enable (gpio_glitch_filter_handle_t filter)**
Enable a glitch filter.

Parameters

- **filter** – [in] Glitch filter handle returned from `gpio_new_flex_glitch_filter` or `gpio_new_pin_glitch_filter`

Returns

- ESP_OK: Enable glitch filter successfully
- ESP_ERR_INVALID_ARG: Enable glitch filter failed because of invalid arguments
- ESP_ERR_INVALID_STATE: Enable glitch filter failed because the glitch filter is already enabled
- ESP_FAIL: Enable glitch filter failed because of other error

**esp_err_t gpio_glitch_filter_disable (gpio_glitch_filter_handle_t filter)**
Disable a glitch filter.

Parameters

- **filter** – [in] Glitch filter handle returned from `gpio_new_flex_glitch_filter` or `gpio_new_pin_glitch_filter`

Returns

- ESP_OK: Disable glitch filter successfully
- ESP_ERR_INVALID_ARG: Disable glitch filter failed because of invalid arguments
- ESP_ERR_INVALID_STATE: Disable glitch filter failed because the glitch filter is not enabled yet
- ESP_FAIL: Disable glitch filter failed because of other error

**Structures**

**struct gpio_pin_glitch_filter_config_t**
Configuration of GPIO pin glitch filter.

**Public Members**

```c

glitch_filter_clock_source_t clk_src
Clock source for the glitch filter

gpio_num_t gpio_num
GPIO number
```
Chapter 2. API Reference

```c
struct gpio_flex_glitch_filter_config_t
    Configuration of GPIO flex glitch filter.

    Public Members

    glitch_filter_clock_source_t clk_src
        Clock source for the glitch filter

    gpio_num_t gpio_num
        GPIO number

    uint32_t window_width_ns
        Sample window width (in ns)

    uint32_t window_thres_ns
        Sample window threshold (in ns), during the window_width_ns sample window, any pulse whose width < window_thres_ns will be discarded.

    Type Definitions

typedef struct gpio_glitch_filter_t *gpio_glitch_filter_handle_t
    Typedef of GPIO glitch filter handle.
```

2.5.7 General Purpose Timer (GPTimer)

Introduction

GPTimer (General Purpose Timer) is the driver of ESP32-S2 Timer Group peripheral. The hardware timer features high resolution and flexible alarm action. The behavior when the internal counter of a timer reaches a specific target value is called a timer alarm. When a timer alarms, a user registered per-timer callback would be called.

Typically, a general purpose timer can be used in scenarios like:

- Free running as a wall clock, fetching a high-resolution timestamp at any time and any places
- Generate period alarms, trigger events periodically
- Generate one-shot alarm, respond in target time

Functional Overview

The following sections of this document cover the typical steps to install and operate a timer:

- **Resource Allocation** - covers which parameters should be set up to get a timer handle and how to recycle the resources when GPTimer finishes working.
- **Set and Get Count Value** - covers how to force the timer counting from a start point and how to get the count value at anytime.
- **Set up Alarm Action** - covers the parameters that should be set up to enable the alarm event.
- **Register Event Callbacks** - covers how to hook user specific code to the alarm event callback function.
- **Enable and Disable Timer** - covers how to enable and disable the timer.
- **Start and Stop Timer** - shows some typical use cases that start the timer with different alarm behavior.
- **Power Management** - describes how different source clock selections can affect power consumption.
• **IRAM Safe** - describes tips on how to make the timer interrupt and IO control functions work better along with a disabled cache.
• **Thread Safety** - lists which APIs are guaranteed to be thread safe by the driver.
• **Kconfig Options** - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

**Resource Allocation**  Different ESP chips might have different numbers of independent timer groups, and within each group, there could also be several independent timers.

A GPTimer instance is represented by `gptimer_handle_t`. The driver behind manages all available hardware resources in a pool, so that you do not need to care about which timer and which group it belongs to.

To install a timer instance, there is a configuration structure that needs to be given in advance: `gptimer_config_t`:

- `gptimer_config_t::clk_src` selects the source clock for the timer. The available clocks are listed in `gptimer_clock_source_t`, you can only pick one of them. For the effect on power consumption of different clock source, please refer to Section **Power Management**.
- `gptimer_config_t::direction` sets the counting direction of the timer, supported directions are listed in `gptimer_count_direction_t`, you can only pick one of them.
- `gptimer_config_t::resolution_hz` sets the resolution of the internal counter. Each count step is equivalent to `1 / resolution_hz` seconds.
- `gptimer_config::intr_priority` sets the priority of the timer interrupt. If it is set to 0, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.
- Optional `gptimer_config_t::intr_shared` sets whether or not mark the timer interrupt source as a shared one. For the pros/cons of a shared interrupt, you can refer to **Interrupt Handling**.

With all the above configurations set in the structure, the structure can be passed to `gptimer_new_timer()` which will instantiate the timer instance and return a handle of the timer.

The function can fail due to various errors such as insufficient memory, invalid arguments, etc. Specifically, when there are no more free timers (i.e., all hardware resources have been used up), then `ESP_ERR_NOT_FOUND` will be returned. The total number of available timers is represented by the `SOC_TIMER_GROUP_TOTAL_TIMERS` and its value depends on the ESP chip.

If a previously created GPTimer instance is no longer required, you should recycle the timer by calling `gptimer_del_timer()`. This allows the underlying HW timer to be used for other purposes. Before deleting a GPTimer handle, please disable it by `gptimer_disable()` in advance or make sure it has not enabled yet by `gptimer_enable()`.

**Creating a GPTimer Handle with Resolution of 1 MHz**

```c
// Create a GPTimer handle with resolution of 1 MHz

gptimer_handle_t gptimer = NULL;
gptimer_config_t timer_config = {
    .clk_src = GPTIMER_CLK_SRC_DEFAULT,
    .direction = GPTIMER_COUNT_UP,
    .resolution_hz = 1 * 1000 * 1000, // 1MHz, 1 tick = 1us
};
ESP_ERROR_CHECK(gptimer_new_timer(&timer_config, &gptimer));
```

**Set and Get Count Value**  When the GPTimer is created, the internal counter will be reset to zero by default. The counter value can be updated asynchronously by `gptimer_set_raw_count()`. The maximum count value is dependent on the bit width of the hardware timer, which is also reflected by the SOC macro `SOC_TIMER_GROUP_COUNTER_BIT_WIDTH`. When updating the raw count of an active timer, the timer will immediately start counting from the new value.

Count value can be retrieved by `gptimer_get_raw_count()`, at any time.

---

1 Different ESP chip series might have different numbers of GPTimer instances. For more details, please refer to **ESP32-S2 Technical Reference Manual > Chapter Timer Group (TIMG) [PDF]**. The driver does forbid you from applying for more timers, but it returns error when all available hardware resources are used up. Please always check the return value when doing resource allocation (e.g., `gptimer_new_timer()`).
**Set up Alarm Action**  For most of the use cases of GPTimer, you should set up the alarm action before starting the timer, except for the simple wall-clock scenario, where a free running timer is enough. To set up the alarm action, you should configure several members of `gptimer_alarm_config_t` based on how you make use of the alarm event:

- `gptimer_alarm_config_t::alarm_count` sets the target count value that triggers the alarm event. You should also take the counting direction into consideration when setting the alarm value. Specially, `gptimer_alarm_config_t::alarm_count` and `gptimer_alarm_config_t::reload_count` cannot be set to the same value when `gptimer_alarm_config_t::auto_reload_on_alarm` is true, as keeping reload with a target alarm count is meaningless.
- `gptimer_alarm_config_t::reload_count` sets the count value to be reloaded when the alarm event happens. This configuration only takes effect when `gptimer_alarm_config_t::auto_reload_on_alarm` is set to true.
- `gptimer_alarm_config_t::auto_reload_on_alarm` flag sets whether to enable the auto-reload feature. If enabled, the hardware timer will reload the value of `gptimer_alarm_config_t::reload_count` into counter immediately when an alarm event happens.

To make the alarm configurations take effect, you should call `gptimer_set_alarm_action()`. Especially, if `gptimer_alarm_config_t` is set to `NULL`, the alarm function will be disabled.

**Note:** If an alarm value is set and the timer has already exceeded this value, the alarm will be triggered immediately.

**Register Event Callbacks**  After the timer starts up, it can generate a specific event (e.g., the “Alarm Event”) dynamically. If you have some functions that should be called when the event happens, please hook your function to the interrupt service routine by calling `gptimer_register_event_callbacks()`. All supported event callbacks are listed in `gptimer_event_callbacks_t`:

- `gptimer_event_callbacks_t::on_alarm` sets a callback function for alarm events. As this function is called within the ISR context, you must ensure that the function does not attempt to block (e.g., by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The function prototype is declared in `gptimer_alarm_cb_t`.

You can save your own context to `gptimer_register_event_callbacks()` as well, via the parameter `user_data`. The user data will be directly passed to the callback function.

This function lazy installs the interrupt service for the timer but not enable it. So please call this function before `gptimer_enable()`, otherwise the `ESP_ERR_INVALID_STATE` error will be returned. See Section Enable and Disable Timer for more information.

**Enable and Disable Timer**  Before doing IO control to the timer, you need to enable the timer first, by calling `gptimer_enable()`. This function:

- Switches the timer driver state from `init` to `enable`.
- Enables the interrupt service if it has been lazy installed by `gptimer_register_event_callbacks()`.
- Acquires a proper power management lock if a specific clock source (e.g., APB clock) is selected. See Section Power Management for more information.

Calling `gptimer_disable()` does the opposite, that is, put the timer driver back to the `init` state, disable the interrupts service and release the power management lock.

**Start and Stop Timer**  The basic IO operation of a timer is to start and stop. Calling `gptimer_start()` can make the internal counter work, while calling `gptimer_stop()` can make the counter stop working. The following illustrates how to start a timer with or without an alarm event.

Calling `gptimer_start()` transits the driver state from `enabled` to `run`, and vice versa. You need to make sure the start and stop functions are used in pairs, otherwise, the functions may return `ESP_ERR_INVALID_STATE`. Most of the time, this error means that the timer is already stopped or in the “start protection” state (i.e., `gptimer_start()` is called but not finished).
Start Timer as a Wall Clock

```c
ESP_ERROR_CHECK(gptimer_enable(gptimer));
ESP_ERROR_CHECK(gptimer_start(gptimer));
// Retrieve the timestamp at any time
uint64_t count;
ESP_ERROR_CHECK(gptimer_get_raw_count(gptimer, &count));
```

Trigger Period Events

```c
typedef struct {
    uint64_t event_count;
} example_queue_element_t;

static bool example_timer_on_alarm_cb(gptimer_handle_t timer, const gptimer_alarm__event_data_t *edata, void *user_ctx)
{
    BaseType_t high_task_awoken = pdFALSE;
    QueueHandle_t queue = (QueueHandle_t)user_ctx;
    // Retrieve the count value from event data
    example_queue_element_t ele = {
        .event_count = edata->count_value
    };
    // Optional: send the event data to other task by OS queue
    // Do not introduce complex logics in callbacks
    // Suggest dealing with event data in the main loop, instead of in this callback
    xQueueSendFromISR(queue, &ele, &high_task_awoken);
    // return whether we need to yield at the end of ISR
    return high_task_awoken == pdTRUE;
}
gptimer_alarm_config_t alarm_config = {
    .reload_count = 0, // counter will reload with 0 on alarm event
    .alarm_count = 1000000, // period = 1s @resolution 1MHz
    .flags.auto_reload_on_alarm = true, // enable auto-reload
};
ESP_ERROR_CHECK(gptimer_set_alarm_action(gptimer, &alarm_config));
gptimer_event_callbacks_t cbs = {
    .on_alarm = example_timer_on_alarm_cb, // register user callback
};
ESP_ERROR_CHECK(gptimer_register_event_callbacks(gptimer, &cbs, queue));
ESP_ERROR_CHECK(gptimer_enable(gptimer));
ESP_ERROR_CHECK(gptimer_start(gptimer));
```

Trigger One-Shot Event

```c
typedef struct {
    uint64_t event_count;
} example_queue_element_t;

static bool example_timer_on_alarm_cb(gptimer_handle_t timer, const gptimer_alarm__event_data_t *edata, void *user_ctx)
{
    BaseType_t high_task_awoken = pdFALSE;
    QueueHandle_t queue = (QueueHandle_t)user_ctx;
    // Stop timer the sooner the better
    gptimer_stop(timer);
    // Retrieve the count value from event data
    example_queue_element_t ele = {
(continues on next page)
Dynamic Alarm Update  Alarm value can be updated dynamically inside the ISR handler callback, by changing \texttt{gptimer\_alarm\_event\_data\_t::alarm\_value}. Then the alarm value will be updated after the callback function returns.

```c
typedef struct {
    uint64_t event_count;
} example_queue_element_t;

static bool example_timer_on_alarm_cb(gptimer_handle_t timer, const gptimer_alarm_event_data_t *edata, void *user_ctx) {
    BaseType_t high_task_awoken = pdFALSE;
    QueueHandle_t queue = (QueueHandle_t)user_data;
    // Retrieve the count value from event data
    example_queue_element_t ele = {
        .event_count = edata->count_value
    };
    // Optional: send the event data to other task by OS queue
    xQueueSendFromISR(queue, &ele, &high_task_awoken);
    // return whether we need to yield at the end of ISR
    return high_task_awoken == pdTRUE;
}

```
**Power Management**  There are some power management strategies, which might turn off or change the frequency of GPTimer’s source clock to save power consumption. For example, during DFS, APB clock will be scaled down. If light-sleep is also enabled, PLL and XTAL clocks will be powered off. Both of them can result in an inaccurate time keeping.

The driver can prevent the above situation from happening by creating different power management lock according to different clock source. The driver increases the reference count of that power management lock in the `gptimer_enable()` and decrease it in the `gptimer_disable()`. So we can ensure the clock source is stable between `gptimer_enable()` and `gptimer_disable()`.

**IRAM Safe**  By default, the GPTimer interrupt will be deferred when the cache is disabled because of writing or erasing the flash. Thus the alarm interrupt will not get executed in time, which is not expected in a real-time application.

There is a Kconfig option `CONFIG_GPTIMER_ISR_IRAM_SAFE` that:

- Enables the interrupt being serviced even when the cache is disabled
- Places all functions that used by the ISR into IRAM
- Places driver object into DRAM (in case it is mapped to PSRAM by accident)

This allows the interrupt to run while the cache is disabled, but comes at the cost of increased IRAM consumption.

There is another Kconfig option `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` that can put commonly used IO control functions into IRAM as well. So, these functions can also be executable when the cache is disabled. These IO control functions are as follows:

- `gptimer_start()`
- `gptimer_stop()`
- `gptimer_get_raw_count()`
- `gptimer_set_raw_count()`
- `gptimer_set_alarm_action()`

**Thread Safety**  All the APIs provided by the driver are guaranteed to be thread safe, which means you can call them from different RTOS tasks without protection by extra locks. The following functions are allowed to run under ISR context.

- `gptimer_start()`
- `gptimer_stop()`
- `gptimer_get_raw_count()`
- `gptimer_set_raw_count()`
- `gptimer_get_captured_count()`
- `gptimer_set_alarm_action()`

**Kconfig Options**

- `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` controls where to place the GPTimer control functions (IRAM or flash).
- `CONFIG_GPTIMER_ISR_HANDLER_IN_IRAM` controls where to place the GPTimer ISR handler (IRAM or flash).
- `CONFIG_GPTIMER_ISR_IRAM_SAFE` controls whether the default ISR handler should be masked when the cache is disabled, see Section IRAM Safe for more information.
- `CONFIG_GPTIMER_ENABLE_DEBUG_LOG` is used to enabled the debug log output. Enable this option will increase the firmware binary size.

**Application Examples**

---

2 `gptimer_event_callbacks_t::on_alarm` callback and the functions invoked by the callback should also be placed in IRAM, please take care of them by yourself.
Typical use cases of GPTimer are listed in the example peripherals/timer_group/gptimer.

API Reference

Header File

- components/esp_driver_gptimer/include/driver/gptimer.h
- This header file can be included with:

```c
#include "driver/gptimer.h"
```

- This header file is a part of the API provided by the esp_driver_gptimer component. To declare that your component depends on esp_driver_gptimer, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_gptimer
```

or

```cmake
PRIV_REQUIRES esp_driver_gptimer
```

Functions

`esp_err_t gptimer_new_timer(const gptimer_config_t *config, gptimer_handle_t *ret_timer)`

Create a new General Purpose Timer, and return the handle.

**Note:** The newly created timer is put in the “init” state.

**Parameters**

- `config` - [in] GPTimer configuration
- `ret_timer` - [out] Returned timer handle

**Returns**

- ESP_OK: Create GPTimer successfully
- ESP.ERR_INVALID_ARG: Create GPTimer failed because of invalid argument
- ESP.ERR_NO_MEM: Create GPTimer failed because out of memory
- ESP.ERR_NOT_FOUND: Create GPTimer failed because all hardware timers are used up and no more free one
- ESP.FAIL: Create GPTimer failed because of other error

`esp_err_t gptimer_del_timer(gptimer_handle_t timer)`

Delete the GPTimer handle.

**Note:** A timer must be in the “init” state before it can be deleted.

**Parameters**

- `timer` - [in] Timer handle created by gptimer_new_timer

**Returns**

- ESP_OK: Delete GPTimer successfully
- ESP.ERR_INVALID_ARG: Delete GPTimer failed because of invalid argument
- ESP.ERR_INVALID_STATE: Delete GPTimer failed because the timer is not in init state
- ESP.FAIL: Delete GPTimer failed because of other error

`esp_err_t gptimer_set_raw_count(gptimer_handle_t timer, uint64_t value)`

Set GPTimer raw count value.
**Note:** When updating the raw count of an active timer, the timer will immediately start counting from the new value.

**Note:** This function is allowed to run within ISR context

**Note:** If CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

### Parameters
- **timer** - [in] Timer handle created by gptimer_new_timer
- **value** - [in] Count value to be set

### Returns
- ESP_OK: Set GPTimer raw count value successfully
- ESP_ERR_INVALID_ARG: Set GPTimer raw count value failed because of invalid argument
- ESP_FAIL: Set GPTimer raw count value failed because of other error

```c
esp_err_t gptimer_get_raw_count(gptimer_handle_t timer, uint64_t*value)
```

Get GPTimer raw count value.

**Note:** This function will trigger a software capture event and then return the captured count value.

**Note:** With the raw count value and the resolution returned from gptimer_get_resolution, you can convert the count value into seconds.

**Note:** This function is allowed to run within ISR context

**Note:** If CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

### Parameters
- **timer** - [in] Timer handle created by gptimer_new_timer
- **value** - [out] Returned GPTimer count value

### Returns
- ESP_OK: Get GPTimer raw count value successfully
- ESP_ERR_INVALID_ARG: Get GPTimer raw count value failed because of invalid argument
- ESP_FAIL: Get GPTimer raw count value failed because of other error

```c
esp_err_t gptimer_get_resolution(gptimer_handle_t timer, uint32_t*out_resolution)
```

Return the real resolution of the timer.

**Note:** usually the timer resolution is same as what you configured in the gptimer_config_t::resolution_hz, but some unstable clock source (e.g. RC_FAST) will do a calibration, the real resolution can be different from the configured one.
Chapter 2. API Reference

Parameters

- **timer** -[in] Timer handle created by `gptimer_new_timer`
- **out_resolution** -[out] Returned timer resolution, in Hz

Returns

- ESP_OK: Get GPTimer resolution successfully
- ESP_ERR_INVALID_ARG: Get GPTimer resolution failed because of invalid argument
- ESP_FAIL: Get GPTimer resolution failed because of other error

```c
esp_err_t gptimer_get_resolution (gptimer_handle_t timer, uint64_t *out_resolution)
```

Get GPTimer resolution.

Note: The capture action can be issued either by ETM event or by software (see also `gptimer_get_raw_count`).

---

Note: This function is allowed to run within ISR context

---

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

---

Parameters

- **timer** -[in] Timer handle created by `gptimer_new_timer`
- **value** -[out] Returned captured count value

Returns

- ESP_OK: Get GPTimer captured count value successfully
- ESP_ERR_INVALID_ARG: Get GPTimer captured count value failed because of invalid argument
- ESP_FAIL: Get GPTimer captured count value failed because of other error

```c
esp_err_t gptimer_get_captured_count (gptimer_handle_t timer, uint64_t *value)
```

Get GPTimer captured count value.

---

Note: The capture action can be issued either by ETM event or by software (see also `gptimer_get_raw_count`).

---

Note: This function is allowed to run within ISR context

---

Note: If `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM` is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

---

Parameters

- **timer** -[in] Timer handle created by `gptimer_new_timer`
- **cbs** -[in] Group of callback functions
- **user_data** -[in] User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set event callbacks failed because the timer is not in init state
- ESP_FAIL: Set event callbacks failed because of other error

```c
esp_err_t gptimer_register_event_callbacks (gptimer_handle_t timer, const gptimer_event_callbacks_t *cbs, void *user_data)
```

Set callbacks for GPTimer.

---

Note: User registered callbacks are expected to be runnable within ISR context

---

Note: The first call to this function needs to be before the call to `gptimer_enable`

---

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

---

Parameters

- **timer** -[in] Timer handle created by `gptimer_new_timer`
- **cbs** -[in] Group of callback functions
- **user_data** -[in] User data, which will be passed to callback functions directly

Returns

- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE: Set event callbacks failed because the timer is not in init state
- ESP_FAIL: Set event callbacks failed because of other error

```c
esp_err_t gptimer_unregister_event_callbacks (gptimer_handle_t timer, const gptimer_event_callbacks_t *cbs, void *user_data)
```

Deregister event callbacks for GPTimer.

---

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

---

Note: The first call to this function needs to be before the call to `gptimer_enable`

---

Note: User registered callbacks are expected to be runnable within ISR context

---

Note: User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.
**esp_err_t gptimer_set_alarm_action** (gptimer_handle_t timer, const gptimer_alarm_config_t *config)

Set alarm event actions for GPTimer.

**Note:** This function is allowed to run within ISR context, so that user can set new alarm action immediately in the ISR callback.

**Note:** If CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

**Parameters**

- **timer** - [in] Timer handle created by gptimer_new_timer
- **config** - [in] Alarm configuration, especially, set config to NULL means disabling the alarm function

**Returns**

- ESP_OK: Set alarm action for GPTimer successfully
- ESP_ERR_INVALID_ARG: Set alarm action for GPTimer failed because of invalid argument
- ESP_FAIL: Set alarm action for GPTimer failed because of other error

**esp_err_t gptimer_enable** (gptimer_handle_t timer)

Enable GPTimer.

**Note:** This function will transit the timer state from “init” to “enable”.

**Note:** This function will enable the interrupt service, if it’s lazy installed in gptimer_register_event_callbacks.

**Note:** This function will acquire a PM lock, if a specific source clock (e.g. APB) is selected in the gptimer_config_t, while CONFIG_PM_ENABLE is enabled.

**Note:** Enable a timer doesn’t mean to start it. See also gptimer_start for how to make the timer start counting.

**Parameters**

- **timer** - [in] Timer handle created by gptimer_new_timer

**Returns**

- ESP_OK: Enable GPTimer successfully
- ESP_ERR_INVALID_ARG: Enable GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable GPTimer failed because the timer is already enabled
- ESP_FAIL: Enable GPTimer failed because of other error

**esp_err_t gptimer_disable** (gptimer_handle_t timer)

Disable GPTimer.

**Note:** This function will transit the timer state from “enable” to “init”.
**Note:** This function will disable the interrupt service if it’s installed.

**Note:** This function will release the PM lock if it’s acquired in the gptimer_enable.

**Note:** Disable a timer doesn’t mean to stop it. See also gptimer_stop for how to make the timer stop counting.

**Parameters**

- **timer** *in* Timer handle created by gptimer_new_timer

**Returns**

- ESP_OK: Disable GPTimer successfully
- ESP_ERR_INVALID_ARG: Disable GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable GPTimer failed because the timer is not enabled yet
- ESP_FAIL: Disable GPTimer failed because of other error

`esp_err_t gptimer_start` *(gptimer_handle_t timer)*

Start GPTimer (internal counter starts counting)

**Note:** This function will transit the timer state from “enable” to “run”.

**Note:** This function is allowed to run within ISR context

**Note:** If CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.

**Parameters**

- **timer** *in* Timer handle created by gptimer_new_timer

**Returns**

- ESP_OK: Start GPTimer successfully
- ESP_ERR_INVALID_ARG: Start GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Start GPTimer failed because the timer is not enabled or is already in running
- ESP_FAIL: Start GPTimer failed because of other error

`esp_err_t gptimer_stop` *(gptimer_handle_t timer)*

Stop GPTimer (internal counter stops counting)

**Note:** This function will transit the timer state from “run” to “enable”.

**Note:** This function is allowed to run within ISR context

**Note:** If CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Flash Cache is disabled.
Chapter 2. API Reference

Returns

- ESP_OK: Stop GPTimer successfully
- ESP_ERR_INVALID_ARG: Stop GPTimer failed because of invalid argument
- ESP_ERR_INVALID_STATE: Stop GPTimer failed because the timer is not in running.
- ESP_FAIL: Stop GPTimer failed because of other error

Structures

struct `gptimer_config_t`

General Purpose Timer configuration.

Public Members

`gptimer_clock_source_t clk_src`

GPTimer clock source

`gptimer_count_direction_t direction`

Count direction

`uint32_t resolution_hz`

Counter resolution (working frequency) in Hz, hence, the step size of each count tick equals to (1 / resolution_hz) seconds

`int intr_priority`

GPTimer interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

`uint32_t intr_shared`

Set true, the timer interrupt number can be shared with other peripherals

struct `gptimer_config_t::[anonymous] flags`

GPTimer config flags

struct `gptimer_event_callbacks_t`

Group of supported GPTimer callbacks.

Note: The callbacks are all running under ISR environment

Note: When CONFIG_GPTIMER_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM.

Public Members

`gptimer_alarm_cb_t on_alarm`

Timer alarm callback

struct `gptimer_alarm_config_t`

General Purpose Timer alarm configuration.
Public Members

```c
uint64_t alarm_count
   Alarm target count value
```

```c
uint64_t reload_count
   Alarm reload count value, effect only when auto_reload_on_alarm is set to true
```

```c
uint32_t auto_reload_on_alarm
   Reload the count value by hardware, immediately at the alarm event
```

```c
struct gptimer_alarm_config_t::*[anonymous] flags
   Alarm config flags
```

Header File

- components/esp_driver_gptimer/include/driver/gptimer_etm.h
- This header file can be included with:
  ```c
  #include "driver/gptimer_etm.h"
  ```
- This header file is a part of the API provided by the esp_driver_gptimer component. To declare that your component depends on esp_driver_gptimer, add the following to your CMakeLists.txt:
  ```cmake
  REQUIRES esp_driver_gptimer
  or
  PRIV_REQUIRES esp_driver_gptimer
  ```

Functions

`esp_err_t gptimer_new_etm_event (gptimer_handle_t timer, const gptimer_etm_event_config_t *config, esp_etm_event_handle_t *out_event)`

Get the ETM event for GPTimer.

**Note:** The created ETM event object can be deleted later by calling `esp_etm_del_event`

**Parameters**

- `timer` - [in] Timer handle created by `gptimer_new_timer`
- `config` - [in] GPTimer ETM event configuration
- `out_event` - [out] Returned ETM event handle

**Returns**

- `ESP_OK`: Get ETM event successfully
- `ESP_ERR_INVALID_ARG`: Get ETM event failed because of invalid argument
- `ESP_FAIL`: Get ETM event failed because of other error

`esp_err_t gptimer_new_etm_task (gptimer_handle_t timer, const gptimer_etm_task_config_t *config, esp_etm_task_handle_t *out_task)`

Get the ETM task for GPTimer.

**Note:** The created ETM task object can be deleted later by calling `esp_etm_del_task`

**Parameters**
• **timer** [in] Timer handle created by `gptimer_new_timer`
• **config** [in] GPTimer ETM task configuration
• **out_task** [out] Returned ETM task handle

**Returns**
- ESP_OK: Get ETM task successfully
- ESP_ERR_INVALID_ARG: Get ETM task failed because of invalid argument
- ESP_FAIL: Get ETM task failed because of other error

**Structures**

struct **gptimer_etm_event_config_t**
GPTimer ETM event configuration.

**Public Members**

- **gptimer_etm_event_type_t** `event_type`
  GPTimer ETM event type

struct **gptimer_etm_task_config_t**
GPTimer ETM task configuration.

**Public Members**

- **gptimer_etm_task_type_t** `task_type`
  GPTimer ETM task type

**Header File**

- components/esp_driver_gptimer/include/driver/gptimer_types.h
- This header file can be included with:

```c
#include "driver/gptimer_types.h"
```

- This header file is a part of the API provided by the `esp_driver_gptimer` component. To declare that your component depends on `esp_driver_gptimer`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_gptimer
```

or

```cmake
PRIV_REQUIRES esp_driver_gptimer
```

**Structures**

struct **gptimer_alarm_event_data_t**
GPTimer alarm event data.

**Public Members**

- **uint64_t** `count_value`
  Current count value
```c
uint64_t alarm_value
    Current alarm value
```

### Type Definitions

typedef struct gptimer_t *gptimer_handle_t
    Type of General Purpose Timer handle.

typedef bool (*gptimer_alarm_cb_t)(gptimer_handle_t timer, const gptimer_alarm_event_data_t *edata, void *user_ctx)
    Timer alarm callback prototype.

    | Param  | Description                                      |
    |--------|--------------------------------------------------|
    | timer  | [in] Timer handle created by gptimer_new_timer   |
    | edata  | [in] Alarm event data, fed by driver             |
    | ctx    | [in] User data, passed from gptimer_register_eventcallbacks |

    Return Whether a high priority task has been waken up by this function

### Header File

- components/hal/include/hal/timer_types.h
- This header file can be included with:

```
#include "hal/timer_types.h"
```

### Type Definitions

typedef soc_periph_gptimer_clk_src_t gptimer_clock_source_t
    GPTimer clock source.

#### Note
User should select the clock source based on the power and resolution requirement

### Enumerations

enum gptimer_count_direction_t
    GPTimer count direction.

    Values:

    Enumerator GPTIMER_COUNT_DOWN
        Decrease count value

    Enumerator GPTIMER_COUNT_UP
        Increase count value

enum gptimer_etm_task_type_t
    GPTimer specific tasks that supported by the ETM module.

    Values:

    Enumerator GPTIMER_ETM_TASK_START_COUNT
        Start the counter
Chapter 2. API Reference

enumerator GPTIMER_ETM_TASK_STOP_COUNT
    Stop the counter

enumerator GPTIMER_ETM_TASK_EN_ALARM
    Enable the alarm

enumerator GPTIMER_ETM_TASK_RELOAD
    Reload preset value into counter

enumerator GPTIMER_ETM_TASK_CAPTURE
    Capture current count value into specific register

enumerator GPTIMER_ETM_TASK_MAX
    Maximum number of tasks

enum gptimer_etm_event_type_t
    GPTimer specific events that supported by the ETM module.
    Values:

enumerator GPTIMER_ETM_EVENT_ALARM_MATCH
    Count value matches the alarm target value

enumerator GPTIMER_ETM_EVENT_MAX
    Maximum number of events

2.5.8 Dedicated GPIO

Overview

The dedicated GPIO is designed for CPU interaction with GPIO matrix and IO MUX. Any GPIO that is configured as “dedicated” can be access by CPU instructions directly, which makes it easy to achieve a high GPIO flip speed, and simulate serial/parallel interface in a bit-banging way. As toggling a GPIO in this “CPU Dedicated” way costs few overhead, it would be great for cases like performance measurement using an oscilloscope.

Create/Destroy GPIO Bundle

A GPIO bundle is a group of GPIOs, which can be manipulated at the same time in one CPU cycle. The maximal number of GPIOs that a bundle can contain is limited by each CPU. What’s more, the GPIO bundle has a strong relevance to the CPU which it derives from. Any operations on the GPIO bundle should be put inside a task which is running on the same CPU core to the GPIO bundle belongs to. Likewise, only those ISRs who are installed on the same CPU core are allowed to do operations on that GPIO bundle.

Note: Dedicated GPIO is more of a CPU peripheral, so it has a strong relationship with CPU core. It’s highly recommended to install and operate GPIO bundle in a pin-to-core task. For example, if GPIOA is connected to CPU0, and the dedicated GPIO instruction is issued from CPU1, then it’s impossible to control GPIOA.
To install a GPIO bundle, one needs to call `dedic_gpio_new_bundle()` to allocate the software resources and connect the dedicated channels to user selected GPIOs. Configurations for a GPIO bundle are covered in `dedic_gpio_bundle_config_t` structure:

- `gpio_array`: An array that contains GPIO number.
- `array_size`: Element number of `gpio_array`.
- `flags`: Extra flags to control the behavior of GPIO Bundle.
  - `in_en` and `out_en` are used to select whether to enable the input and output function (note, they can be enabled together).
  - `in_invert` and `out_invert` are used to select whether to invert the GPIO signal.

The following code shows how to install a output only GPIO bundle:

```c
// configure GPIO
const int bundleA_gpios[] = {0, 1};
gpio_config_t io_conf = { 
  .mode = GPIO_MODE_OUTPUT,
};
for (int i = 0; i < sizeof(bundleA_gpios) / sizeof(bundleA_gpios[0]); i++) { 
  io_conf.pin_bit_mask = 1ULL << bundleA_gpios[i];
  gpio_config(&io_conf);
}

// Create bundleA, output only
dedic_gpio_bundle_handle_t bundleA = NULL;
dedic_gpio_bundle_config_t bundleA_config = { 
  .gpio_array = bundleA_gpios,
  .array_size = sizeof(bundleA_gpios) / sizeof(bundleA_gpios[0]),
  .flags = {
    .out_en = 1,
  },
};
ESP_ERROR_CHECK(dedic_gpio_new_bundle(&bundleA_config, &bundleA));
```

To uninstall the GPIO bundle, one needs to call `dedic_gpio_del_bundle()`.

**Note:** `dedic_gpio_new_bundle()` doesn’t cover any GPIO pad configuration (e.g., pull up/down, drive ability, output/input enable), so before installing a dedicated GPIO bundle, you have to configure the GPIO separately using GPIO driver API (e.g., `gpio_config()`). For more information about GPIO driver, please refer to [GPIO API Reference](#).

## GPIO Bundle Operations

<table>
<thead>
<tr>
<th>Operations</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write to GPIOs in the bundle by mask</td>
<td><code>dedic_gpio_bundle_write()</code></td>
</tr>
<tr>
<td>Read the value that output from the given GPIO bundle</td>
<td><code>dedic_gpio_bundle_read_out()</code></td>
</tr>
<tr>
<td>Read the value that input to the given GPIO bundle</td>
<td><code>dedic_gpio_bundle_read_in()</code></td>
</tr>
</tbody>
</table>

**Note:** Using the above functions might not get a high GPIO flip speed because of the overhead of function calls and the bit operations involved inside. Users can try [Manipulate GPIOs by Writing Assembly Code](#) instead to reduce the overhead but should take care of the thread safety by themselves.

## Manipulate GPIOs by Writing Assembly Code

For advanced users, they can always manipulate the GPIOs by writing assembly code or invoking CPU Low Level APIs. The usual procedure could be:
1. Allocate a GPIO bundle: `dedic_gpio_new_bundle()`
2. Query the mask occupied by that bundle: `dedic_gpio_get_out_mask()` or `dedic_gpio_get_in_mask()`.
3. Call CPU LL apis (e.g., `dedic_gpio_cpu_ll_write_mask`) or write assembly code with that mask.
4. The fastest way of toggling IO is to use the dedicated “set/clear” instructions:
   - Set bits of GPIO: `set_bit_gpio_out imm[7:0]`
   - Clear bits of GPIO: `clr_bit_gpio_out imm[7:0]`
   - Note: Immediate value width depends on the number of dedicated GPIO channels.

For details of supported dedicated GPIO instructions, please refer to ESP32-S2 Technical Reference Manual > IO MUX and GPIO Matrix (GPIO, IO_MUX) [PDF].

Some of the dedicated CPU instructions are also wrapped inside `hal/dedic_gpio_cpu_ll.h` as helper inline functions.

**Note:** Writing assembly code in application could make your code hard to port between targets, because those customized instructions are not guaranteed to remain the same format on different targets.

**Interrupt Handling**

Dedicated GPIO can also trigger interrupt on specific input event. All supported events are defined in `dedic_gpio_intr_type_t`.

One can enable and register interrupt callback by calling `dedic_gpio_bundle_set_interrupt_and_callback()`. The prototype of the callback function is defined in `dedic_gpio_isr_callback_t`. Keep in mind, the callback should return true if there’s some high priority task woken up.

```c
// user defined ISR callback
IRAM_ATTR bool dedic_gpio_isr_callback(dedic_gpio_bundle_handle_t bundle, uint32_t index, void *args)
{
    SemaphoreHandle_t sem = (SemaphoreHandle_t)args;
    BaseType_t high_task_wakeup = pdFALSE;
    xSemaphoreGiveFromISR(sem, &high_task_wakeup);
    return high_task_wakeup = pdTRUE;
}

// enable positive edge interrupt on the second GPIO in the bundle (i.e., index 1)
ESP_ERROR_CHECK(dedic_gpio_bundle_set_interrupt_and_callback(bundle, BIT(1), DEDIC_GPIO_INTR_POS_EDGE, dedic_gpio_isr_callback, sem));

// wait for done semaphore
xSemaphoreTake(sem, portMAX_DELAY);
```

**Application Example**

Matrix keyboard example based on dedicated GPIO: `peripherals/gpio/matrix_keyboard`.

**API Reference**

**Header File**

- `components/esp_driver_gpio/include/driver/dedic_gpio.h`
- This header file can be included with:

```
#include "driver/dedic_gpio.h"
```
Chapter 2. API Reference

- This header file is a part of the API provided by the `esp_driver_gpio` component. To declare that your component depends on `esp_driver_gpio`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_gpio
```

or

```cmake
PRIV_REQUIRES esp_driver_gpio
```

**Functions**

`esp_err_t dedi_gpio_get_out_mask(dedic_gpio_bundle_handle_t bundle, uint32_t*mask)`

Get allocated channel mask.

**Note:** Each bundle should have at least one mask (in or/and out), based on bundle configuration.

**Note:** With the returned mask, user can directly invoke LL function like ‘`dedic_gpio_cpu_ll_write_mask`’ or write assembly code with dedicated GPIO instructions, to get better performance on GPIO manipulation.

**Parameters**

- **bundle** [in] Handle of GPIO bundle that returned from ‘`dedic_gpio_new_bundle`’
- **mask** [out] Returned mask value for on specific direction (in or out)

**Returns**

- ESP_OK: Get channel mask successfully
- ESP_ERR_INVALID_ARG: Get channel mask failed because of invalid argument
- ESP_FAIL: Get channel mask failed because of other error

`esp_err_t dedi_gpio_get_in_mask(dedic_gpio_bundle_handle_t bundle, uint32_t*mask)`

`esp_err_t dedi_gpio_get_out_offset(dedic_gpio_bundle_handle_t bundle, uint32_t*offset)`

Get the channel offset of the GPIO bundle.

A GPIO bundle maps the GPIOS of a particular direction to a consecutive set of channels within a particular GPIO bank of a particular CPU. This function returns the offset to the bundle’s first channel of a particular direction within the bank.

**Parameters**

- **bundle** [in] Handle of GPIO bundle that returned from ‘`dedic_gpio_new_bundle`’
- **offset** [out] Offset value to the first channel of a specific direction (in or out)

**Returns**

- ESP_OK: Get channel offset successfully
- ESP_ERR_INVALID_ARG: Get channel offset failed because of invalid argument
- ESP_FAIL: Get channel offset failed because of other error

`esp_err_t dedi_gpio_get_in_offset(dedic_gpio_bundle_handle_t bundle, uint32_t*offset)`

`esp_err_t dedi_gpio_new_bundle(const dedic_gpio_bundle_config_t*config, dedic_gpio_bundle_handle_t*ret_bundle)`

Create GPIO bundle and return the handle.

**Note:** One has to enable at least input or output mode in ‘`config`’ parameter.

**Parameters**

- **config** [in] Configuration of GPIO bundle
- **ret_bundle** [out] Returned handle of the new created GPIO bundle

**Returns**
• ESP_OK: Create GPIO bundle successfully  
• ESP_ERR_INVALID_ARG: Create GPIO bundle failed because of invalid argument  
• ESP_ERR_NO_MEM: Create GPIO bundle failed because of no capable memory  
• ESP_ERR_NOT_FOUND: Create GPIO bundle failed because of no enough continuous dedicated channels  
• ESP_FAIL: Create GPIO bundle failed because of other error

```c
esp_err_t dedic_gpio_del_bundle(dedic_gpio_bundle_handle_t bundle)

Destroy GPIO bundle.

Parameters
  bundle [in] Handle of GPIO bundle that returned from “dedic_gpio_new_bundle”

Returns
  • ESP_OK: Destroy GPIO bundle successfully  
  • ESP_ERR_INVALID_ARG: Destroy GPIO bundle failed because of invalid argument  
  • ESP_FAIL: Destroy GPIO bundle failed because of other error
```

```c
void dedic_gpio_bundle_write(dedic_gpio_bundle_handle_t bundle, uint32_t mask, uint32_t value)

Write value to GPIO bundle.

Note: The mask is seen from the view of GPIO bundle. For example, bundleA contains [GPIO10, GPIO12, GPIO17], to set GPIO17 individually, the mask should be 0x04.

Note: For performance reasons, this function doesn’t check the validity of any parameters, and is placed in IRAM.
```

```c
uint32_t dedic_gpio_bundle_read_out(dedic_gpio_bundle_handle_t bundle)

Read the value that output from the given GPIO bundle.

Note: For performance reasons, this function doesn’t check the validity of any parameters, and is placed in IRAM.
```

```c
uint32_t dedic_gpio_bundle_read_in(dedic_gpio_bundle_handle_t bundle)

Read the value that input to the given GPIO bundle.

Note: For performance reasons, this function doesn’t check the validity of any parameters, and is placed in IRAM.
```
\texttt{esp_err_t} \texttt{dedic_gpio_bundle_set_interrupt_and_callback} \( (\text{dedic_gpio_bundle_handle_t} \texttt{bundle}, \)
\( \text{uint32_t} \texttt{mask}, \)
\( \text{dedic_gpio_intr_type_t} \texttt{intr_type}, \)
\( \text{dedic_gpio_isr_callback_t} \texttt{cb_isr}, \)
\( \text{void} *\texttt{cb_args} ) \)

Set interrupt and callback function for GPIO bundle.

**Note:** This function is only valid for bundle with input mode enabled. See “dedic_gpio_bundle_config_t”

**Note:** The mask is seen from the view of GPIO Bundle. For example, bundleA contains [GPIO10, GPIO12, GPIO17], to set GPIO17 individually, the mask should be 0x04.

**Parameters**
- \texttt{bundle} – [in] Handle of GPIO bundle that returned from “dedic_gpio_new_bundle”
- \texttt{mask} – [in] Mask of the GPIOs in the given bundle
- \texttt{intr_type} – [in] Interrupt type, set to DEDIC_GPIO_INTR_NONE can disable interrupt
- \texttt{cb_isr} – [in] Callback function, which got invoked in ISR context. A NULL pointer here will bypass the callback
- \texttt{cb_args} – [in] User defined argument to be passed to the callback function

**Returns**
- ESP_OK: Set GPIO interrupt and callback function successfully
- ESP_ERR_INVALID_ARG: Set GPIO interrupt and callback function failed because of invalid argument
- ESP_FAIL: Set GPIO interrupt and callback function failed because of other error

**Structures**

\textbf{struct dedic_gpio_bundle_config_t}

Type of Dedicated GPIO bundle configuration.

**Public Members**

\textbf{const int *\texttt{gpio_array}}

Array of GPIO numbers, gpio_array[0] \~\~\~ gpio_array[size-1] \leftrightarrow low_dedic_channel_num \~\~\~ high_dedic_channel_num

\textbf{size_t \texttt{array_size}}

Number of GPIOs in gpio_array

\textbf{unsigned int \texttt{in_en}}

Enable input

\textbf{unsigned int \texttt{in_invert}}

Invert input signal

\textbf{unsigned int \texttt{out_en}}

Enable output
unsigned int out_invert
    Invert output signal

struct dedic_gpio_bundle_config_t::[anonymous] flags
    Flags to control specific behaviour of GPIO bundle

**Type Definitions**

typedef struct dedic_gpio_bundle_t *dedic_gpio_bundle_handle_t
    Type of Dedicated GPIO bundle.

typedef bool (*dedic_gpio_isr_callback_t)(dedic_gpio_bundle_handle_t bundle, uint32_t index, void *args)
    Type of dedicated GPIO ISR callback function.

    **Param bundle** Handle of GPIO bundle that returned from “dedic_gpio_new_bundle”
    **Param index** Index of the GPIO in its corresponding bundle (count from 0)
    **Param args** User defined arguments for the callback function. It’s passed through
    dedic_gpio_bundle_set_interrupt_and_callback

    **Return** If a high priority task is woken up by the callback function

**Enumerations**

enum dedic_gpio_intr_type_t
    Supported type of dedicated GPIO interrupt.

    **Values:**

    enumerator DEDIC_GPIO_INTR_NONE
        No interrupt
    enumerator DEDIC_GPIO_INTR_LOW_LEVEL
        Interrupt on low level
    enumerator DEDIC_GPIO_INTR_HIGH_LEVEL
        Interrupt on high level
    enumerator DEDIC_GPIO_INTR_NEG_EDGE
        Interrupt on negedge
    enumerator DEDIC_GPIO_INTR_POS_EDGE
        Interrupt on posedge
    enumerator DEDIC_GPIO_INTR_BOTH_EDGE
        Interrupt on both negedge and posedge

**2.5.9 Hash-Based Message Authentication Code (HMAC)**

Hash-based Message Authentication Code (HMAC) is a secure authentication technique that verifies the authenticity and integrity of a message with a pre-shared key. This module provides hardware acceleration for SHA256-HMAC generation using a key burned into an eFuse block.
For more detailed information on the application workflow and the HMAC calculation process, see ESP32-S2 Technical Reference Manual > HMAC Accelerator (HMAC) [PDF].

**Generalized Application Scheme**

Let there be two parties, A and B. They want to verify the authenticity and integrity of messages sent between each other. Before they can start sending messages, they need to exchange the secret key via a secure channel.

To verify A’s messages, B can do the following:

- A calculates the HMAC of the message it wants to send.
- A sends the message and the HMAC to B.
- B calculates the HMAC of the received message itself.
- B checks whether the received and calculated HMACs match.

If they do match, the message is authentic.

However, the HMAC itself is not bound to this use case. It can also be used for challenge-response protocols supporting HMAC or as a key input for further security modules (see below), etc.

**HMAC on ESP32-S2**

On ESP32-S2, the HMAC module works with a secret key burnt into the eFuses. This eFuse key can be made completely inaccessible for any resources outside the cryptographic modules, thus avoiding key leakage.

Furthermore, ESP32-S2 has three different application scenarios for its HMAC module:

1. HMAC is generated for software use
2. HMAC is used as a key for the Digital Signature (DS) module
3. HMAC is used for enabling the soft-disabled JTAG interface

The first mode is called **Upstream** mode, while the last two modes are called **Downstream** modes.

**eFuse Keys for HMAC**  Six physical eFuse blocks can be used as keys for the HMAC module: block 4 ~ block 9. The enum `hmac_key_id_t` in the API maps them to HMAC_KEY0 ~ HMAC_KEY5.

Each key has a corresponding eFuse parameter **key purpose** determining for which of the three HMAC application scenarios (see below) the key may be used:

<table>
<thead>
<tr>
<th>Key Purpose</th>
<th>Application Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>HMAC generated for software use</td>
</tr>
<tr>
<td>7</td>
<td>HMAC used as a key for the Digital Signature (DS) module</td>
</tr>
<tr>
<td>6</td>
<td>HMAC used for enabling the soft-disabled JTAG interface</td>
</tr>
<tr>
<td>5</td>
<td>HMAC both as a key for the DS module and for enabling JTAG</td>
</tr>
</tbody>
</table>

This is to prevent the usage of a key for a different function than originally intended.

To calculate an HMAC, the software has to provide the ID of the key block containing the secret key as well as the **key purpose** (see ESP32-S2 Technical Reference Manual > eFuse Controller (eFuse) [PDF]).

Before the HMAC key calculation, the HMAC module looks up the purpose of the provided key block. The calculation only proceeds if the purpose of the provided key block matches the purpose stored in the eFuses of the key block provided by the ID.

**HMAC Generation for Software**  Key purpose value: 8

In this case, the HMAC is given out to the software, e.g., to authenticate a message.

The API to calculate the HMAC is `esp_hmac_calculate()`. The input arguments for the function are the message, message length, and the eFuse key block ID which contains the secret and has the eFuse key purpose set to Upstream mode.
HMAC for Digital Signature  Key purpose values: 7, 5

The HMAC can be used as a key derivation function to decrypt private key parameters which are used by the Digital Signature module. A standard message is used by the hardware in that case. You only need to provide the eFuse key block and purpose on the HMAC side, additional parameters are required for the Digital Signature component in that case.

Neither the key nor the actual HMAC is ever exposed outside the HMAC module and DS component. The calculation of the HMAC and its handover to the DS component happen internally.

For more details, see ESP32-S2 Technical Reference Manual > Digital Signature (DS) [PDF].

HMAC for Enabling JTAG  Key purpose values: 6, 5

The third application is using the HMAC as a key to enable JTAG if it was soft-disabled before.

Following is the procedure to re-enable the JTAG:

**Stage 1: Setup**

1. Generate a 256-bit HMAC secret key to use for JTAG re-enable.
2. Write the key to an eFuse block with key purpose HMAC_DOWN_ALL (5) or HMAC_DOWN_JTAG (6). This can be done using the esp_efuse_write_key() function in the firmware or using espefuse.py from the host.
3. Configure the eFuse key block to be read-protected using the esp_efuse_set_read_protect(), so that software cannot read back the value.
4. Burn the soft JTAG disable bit/bits on ESP32-S2. This will permanently disable JTAG unless the correct key value is provided by the software.

**Note:** The API esp_efuse_write_field_bit(ESP_EFUSE_SOFT_DIS_JTAG) can be used to burn soft JTAG disable bit on ESP32-S2.

**Note:** If HARD_DIS_JTAG eFuse is set, then SOFT_DIS_JTAG functionality does not work because JTAG is permanently disabled.

JTAG enables

1. The key to re-enable JTAG is the output of the HMAC-SHA256 function using the secret key in eFuse and 32 0x00 bytes as the message.
2. Pass this key value when calling the esp_hmac_jtag_enable() function from the firmware.
3. To re-disable JTAG in the firmware, reset the system or call esp_hmac_jtag_disable().

For more details, see ESP32-S2 Technical Reference Manual > HMAC Accelerator (HMAC) [PDF].

**Application Outline**

The following code is an outline of how to set an eFuse key and then use it to calculate an HMAC for software usage.

We use esp_efuse_write_key to set physical key block 4 in the eFuse for the HMAC module together with its purpose. ESP_EFUSE_KEY_PURPOSE_HMAC_UP (8) means that this key can only be used for HMAC generation for software usage:

```c
#include "esp_efuse.h"

const uint8_t key_data[32] = { ... };

esp_err_t status = esp_efuse_write_key(EFUSE_BLK_KEY4, ESP_EFUSE_KEY_PURPOSE_HMAC_UP, key_data, sizeof(key_data));
```

(continues on next page)
if (status == ESP_OK) {
    // written key
} else {
    // writing key failed, maybe written already
}

Now we can use the saved key to calculate an HMAC for software usage.

```c
#include "esp_hmac.h"

uint8_t hmac[32];

const char *message = "Hello, HMAC!";
const size_t msg_len = 12;

esp_err_t result = esp_hmac_calculate(HMAC_KEY4, message, msg_len, hmac);

if (result == ESP_OK) {
    // HMAC written to hmac now
} else {
    // failure calculating HMAC
}
```

**API Reference**

**Header File**

- components/esp_hw_support/include/esp_hmac.h
- This header file can be included with:

  ```c
  #include "esp_hmac.h"
  ```

**Functions**

`esp_err_t esp_hmac_calculate (hmac_key_id_t key_id, const void *message, size_t message_len, uint8_t *hmac)`

Calculate the HMAC of a given message.

Calculate the HMAC `hmac` of a given message `message` with length `message_len`. SHA256 is used for the calculation.

**Note:** Uses the HMAC peripheral in “upstream” mode.

**Parameters**

- `key_id` - Determines which of the 6 key blocks in the efuses should be used for the HMAC calculation. The corresponding purpose field of the key block in the efuse must be set to the HMAC upstream purpose value.
- `message` - the message for which to calculate the HMAC
- `message_len` - message length return ESP_ERR_INVALID_STATE if unsuccessful
- `hmac` - [out] the hmac result; the buffer behind the provided pointer must be a writeable buffer of 32 bytes

**Returns**

- ESP_OK, if the calculation was successful,
- ESP_ERR_INVALID_ARG if message or hmac is a nullptr or if key_id out of range
- ESP_FAIL, if the hmac calculation failed
**esp_err_t esp_hmac_jtag_enable (hmac_key_id_t key_id, const uint8_t *token)**

Use HMAC peripheral in Downstream mode to re-enable the JTAG, if it is not permanently disabled by HW. In downstream mode, HMAC calculations performed by peripheral are used internally and not provided back to user.

**Note:** Return value of the API does not indicate the JTAG status.

**Parameters**

• **key_id** – Determines which of the 6 key blocks in the efuses should be used for the HMAC calculation. The corresponding purpose field of the key block in the efuse must be set to HMAC downstream purpose.

• **token** – Pre-calculated HMAC value of the 32-byte 0x00 using SHA-256 and the known private HMAC key. The key is already programmed to a eFuse key block. The key block number is provided as the first parameter to this function.

**Returns**

• **ESP_OK**, if the key_purpose of the key_id matches to HMAC downstream mode, The API returns success even if calculated HMAC does not match with the provided token. However, The JTAG will be re-enabled only if the calculated HMAC value matches with provided token, otherwise JTAG will remain disabled.

• **ESP_FAIL**, if the key_purpose of the key_id is not set to HMAC downstream purpose or JTAG is permanently disabled by EFUSE_HARD_DIS_JTAG eFuse parameter.

• **ESP_ERR_INVALID_ARG**, invalid input arguments

**esp_err_t esp_hmac_jtag_disable (void)**

Disable the JTAG which might be enabled using the HMAC downstream mode. This function just clears the result generated by calling esp_hmac_jtag_enable() API.

**Returns**

• **ESP_OK** return ESP_OK after writing the HMAC_SET_INVALIDATE_JTAG_REG with value 1.

**Enumerations**

enum **hmac_key_id_t**

The possible efuse keys for the HMAC peripheral

**Values:**

enumerator **HMAC_KEY0**

enumerator **HMAC_KEY1**

enumerator **HMAC_KEY2**

enumerator **HMAC_KEY3**

enumerator **HMAC_KEY4**

enumerator **HMAC_KEY5**

enumerator **HMAC_KEY_MAX**
2.5.10 Digital Signature (DS)

The Digital Signature (DS) module provides hardware acceleration of signing messages based on RSA. It uses pre-encrypted parameters to calculate a signature. The parameters are encrypted using HMAC as a key-derivation function. In turn, the HMAC uses eFuses as the input key. The whole process happens in hardware so that neither the decryption key for the RSA parameters nor the input key for the HMAC key derivation function can be seen by the software while calculating the signature.

For more detailed information on the hardware involved in the signature calculation and the registers used, see ESP32-S2 Technical Reference Manual > Digital Signature (DS) [PDF].

Private Key Parameters

The private key parameters for the RSA signature are stored in flash. To prevent unauthorized access, they are AES-encrypted. The HMAC module is used as a key-derivation function to calculate the AES encryption key for the private key parameters. In turn, the HMAC module uses a key from the eFuses key block which can be read-protected to prevent unauthorized access as well.

Upon signature calculation invocation, the software only specifies which eFuse key to use, the corresponding eFuse key purpose, the location of the encrypted RSA parameters, and the message.

Key Generation

Both the HMAC key and the RSA private key have to be created and stored before the DS peripheral can be used. This needs to be done in software on the ESP32-S2 or alternatively on a host. For this context, ESP-IDF provides `esp_efuse_write_block()` to set the HMAC key and `esp_hmac_calculate()` to encrypt the private RSA key parameters.

You can find instructions on how to calculate and assemble the private key parameters in ESP32-S2 Technical Reference Manual > Digital Signature (DS) [PDF].

Signature Calculation with ESP-IDF

For more detailed information on the workflow and the registers used, see ESP32-S2 Technical Reference Manual > Digital Signature (DS) [PDF].

Three parameters need to be prepared to calculate the digital signature:

1. The eFuse key block ID which is used as the key for the HMAC
2. The location of the encrypted private key parameters
3. The message to be signed

Since the signature calculation takes some time, there are two possible API versions to use in ESP-IDF. The first one is `esp_ds_sign()` and simply blocks until the calculation is finished. If software needs to do something else during the calculation, `esp_ds_start_sign()` can be called, followed by periodic calls to `esp_ds_is_busy()` to check when the calculation has finished. Once the calculation has finished, `esp_ds_finish_sign()` can be called to get the resulting signature.

The APIs `esp_ds_sign()` and `esp_ds_start_sign()` calculate a plain RSA signature with the help of the DS peripheral. This signature needs to be converted to an appropriate format for further use. For example, the MbedTLS SSL stack supports PKCS#1 format. The API `esp_ds_rsa_sign()` can be used to obtain the signature directly in the PKCS#1 v1.5 format. It internally uses `esp_ds_start_sign()` and converts the signature into PKCS#1 v1.5 format.

**Note:** This is only the basic DS building block, the message length is fixed. To create signatures of arbitrary messages, the input is normally a hash of the actual message, padded up to the required length. An API to do this is
planned in the future.

Configure the DS Peripheral for a TLS Connection

The DS peripheral on ESP32-S2 chip must be configured before it can be used for a TLS connection. The configuration involves the following steps:

1) Randomly generate a 256-bit value called the **Initialization Vector (IV)**.
2) Randomly generate a 256-bit value called the **HMAC_KEY**.
3) Calculate the encrypted private key parameters from the client private key (RSA) and the parameters generated in the above steps.
4) Then burn the 256-bit **HMAC_KEY** on the eFuse, which can only be read by the DS peripheral.


To configure the DS peripheral for development purposes, you can use the `esp-secure-cert-tool`.

The encrypted private key parameters obtained after the DS peripheral configuration are then to be kept in flash. Furthermore, they are to be passed to the DS peripheral which makes use of those parameters for the Digital Signature operation. The application then needs to read the DS data from the flash, which has been done through the APIs provided by the `esp_secure_cert_mgr` component. Please refer to the component/README for more details.

The process of initializing the DS peripheral and then performing the Digital Signature operation is done internally with the help of ESP-TLS. Please refer to [Digital Signature with ESP-TLS](https://www.espressif.com/cn/technical-reference-manual/esp32-s2#digital-signature-esp-tls) for more details.

As mentioned in the ESP-TLS documentation, the application only needs to provide the encrypted private key parameters to the `esp_tls` context (as `ds_data`), which internally performs all necessary operations for initializing the DS peripheral and then performing the DS operation.

Example for SSL Mutual Authentication Using DS

The example `protocols/mqtt/ssl_ds` shows how to use the DS peripheral for mutual authentication. The example uses `mqtt_client` (Implemented through ESP-MQTT) to connect to broker `test.mosquitto.org` using SSL transport with mutual authentication. The SSL part is internally performed with ESP-TLS. See `protocols/mqtt/ssl_ds/README.md` for more details.

**API Reference**

**Header File**

- `components/esp_hw_support/include/esp_ds.h`
- This header file can be included with:

  ```c
  #include "esp_ds.h"
  ```

**Functions**

```c
esp_err_t esp_ds_sign(const void *message, const esp_ds_data_t *data, hmac_key_id_t key_id, void *signature)
```

Sign the message with a hardware key from specific key slot. The function calculates a plain RSA signature with help of the DS peripheral. The RSA encryption operation is as follows: \( Z = XY \mod M \) where, \( Z \) is the signature, \( X \) is the input message, \( Y \) and \( M \) are the RSA private key parameters.

This function is a wrapper around `esp_ds_finish_sign()` and `esp_ds_start_sign()`, so do not use them in parallel. It blocks until the signing is finished and then returns the signature.

**Note:** Please see note section of `esp_ds_start_sign()` for more details about the input parameters.

Submit Document Feedback
Parameters

- **message** – the message to be signed; its length should be \((data->rsa_length + 1)*4\) bytes, and those bytes must be in little endian format. It is your responsibility to apply your hash function and padding before calling this function, if required. (e.g. message = padding(hash(inputMsg)))
- **data** – the encrypted signing key data (AES encrypted RSA key + IV)
- **key_id** – the HMAC key ID determining the HMAC key of the HMAC which will be used to decrypt the signing key data
- **signature** – the destination of the signature, should be \((data->rsa_length + 1)*4\) bytes long

Returns

- **ESP_OK** if successful, the signature was written to the parameter `signature`.
- **ESP_ERR_INVALID_ARG** if one of the parameters is NULL or `data->rsa_length` is too long or 0
- **ESP_ERR_HW_CRYPTO_DS_HMAC_FAIL** if there was an HMAC failure during retrieval of the decryption key
- **ESP_ERR_NO_MEM** if there hasn’t been enough memory to allocate the context object
- **ESP_ERR_HW_CRYPTO_DS_INVALID_KEY** if there’s a problem with passing the HMAC key to the DS component
- **ESP_ERR_HW_CRYPTO_DS_INVALID_DIGEST** if the message digest didn’t match; the signature is invalid.
- **ESP_ERR_HW_CRYPTO_DS_INVALID_PADDING** if the message padding is incorrect, the signature can be read though since the message digest matches.

```c
esp_err_t esp_ds_start_sign(const void *message, const esp_ds_data_t *data, hmac_key_id_t key_id, esp_ds_context_t **esp_ds_ctx)
```

Start the signing process.

This function yields a context object which needs to be passed to `esp_ds_finish_sign()` to finish the signing process. The function calculates a plain RSA signature with help of the DS peripheral. The RSA encryption operation is as follows: \(Z = XY \mod M\) where, \(Z\) is the signature, \(X\) is the input message, \(Y\) and \(M\) are the RSA private key parameters.

**Note:** This function locks the HMAC, SHA, AES and RSA components, so the user has to ensure to call `esp_ds_finish_sign()` in a timely manner. The numbers \(Y, M, Rb\) which are a part of `esp_ds_data_t` should be provided in little endian format and should be of length equal to the RSA private key bit length. The message length in bytes should also be equal to the RSA private key bit length. No padding is applied to the message automatically. Please ensure the message is appropriate padded before calling the API.

Parameters

- **message** – the message to be signed; its length should be \((data->rsa_length + 1)*4\) bytes, and those bytes must be in little endian format. It is your responsibility to apply your hash function and padding before calling this function, if required. (e.g. message = padding(hash(inputMsg)))
- **data** – the encrypted signing key data (AES encrypted RSA key + IV)
- **key_id** – the HMAC key ID determining the HMAC key of the HMAC which will be used to decrypt the signing key data
- **esp_ds_ctx** – the context object which is needed for finishing the signing process later

Returns

- **ESP_OK** if successful, the ds operation was started now and has to be finished with `esp_ds_finish_sign()`
- **ESP_ERR_INVALID_ARG** if one of the parameters is NULL or `data->rsa_length` is too long or 0
- **ESP_ERR_HW_CRYPTO_DS_HMAC_FAIL** if there was an HMAC failure during retrieval of the decryption key
- **ESP_ERR_NO_MEM** if there hasn’t been enough memory to allocate the context object
- **ESP_ERR_HW_CRYPTO_DS_INVALID_KEY** if there’s a problem with passing the
HMAC key to the DS component

bool esp_ds_is_busy (void)

Return true if the DS peripheral is busy, otherwise false.

Note: Only valid if esp_ds_start_sign() was called before.

esp_err_t esp_ds_finish_sign (void *signature, esp_ds_context_t *esp_ds_ctx)

Finish the signing process.

Parameters

- **signature** – the destination of the signature, should be (data->rsa_length + 1)*4 bytes long, the resultant signature bytes shall be written in little endian format.
- **esp_ds_ctx** – the context object retrieved by esp_ds_start_sign()

Returns

- ESP_OK if successful, the ds operation has been finished and the result is written to signature.
- ESP_ERR_INVALID_ARG if one of the parameters is NULL
- ESP_ERR_HW_CRYPTO_DS_INVALID_DIGEST if the message digest didn’t match; the signature is invalid. This means that the encrypted RSA key parameters are invalid, indicating that they may have been tampered with or indicating a flash error, etc.
- ESP_ERR_HW_CRYPTO_DS_INVALID_PADDING if the message padding is incorrect, the signature can be read though since the message digest matches (see TRM for more details).

esp_err_t esp_ds_encrypt_params (esp_ds_data_t *data, const void *iv, const esp_ds_p_data_t *p_data, const void *key)

Encrypt the private key parameters.

The encryption is a prerequisite step before any signature operation can be done. It is not strictly necessary to use this encryption function, the encryption could also happen on an external device.

Note: The numbers Y, M, Rb which are a part of esp_ds_data_t should be provided in little endian format and should be of length equal to the RSA private key bit length. The message length in bits should also be equal to the RSA private key bit length. No padding is applied to the message automatically. Please ensure the message is appropriate padded before calling the API.

Parameters

- **data** – Output buffer to store encrypted data, suitable for later use generating signatures.
- **iv** – Pointer to 16 byte IV buffer, will be copied into ‘data’. Should be randomly generated bytes each time.
- **p_data** – Pointer to input plaintext key data. The expectation is this data will be deleted after this process is done and ‘data’ is stored.
- **key** – Pointer to 32 bytes of key data. Type determined by key_type parameter. The expectation is the corresponding HMAC key will be stored to efuse and then permanently erased.

Returns

- ESP_OK if successful, the ds operation has been finished and the result is written to signature.
- ESP_ERR_INVALID_ARG if one of the parameters is NULL or p_data->rsa_length is too long

Structures

struct esp_digital_signature_data

Encrypted private key data. Recommended to store in flash in this format.
Note: This struct has to match to one from the ROM code! This documentation is mostly taken from there.

Public Members

**esp_digital_signature_length_t rsa_length**

RSA LENGTH register parameters (number of words in RSA key & operands, minus one).

This value must match the length field encrypted and stored in ‘c’, or invalid results will be returned.
(The DS peripheral will always use the value in ‘c’, not this value, so an attacker can’t alter the DS peripheral results this way, it will just truncate or extend the message and the resulting signature in software.)

Note: In IDF, the enum type length is the same as of type unsigned, so they can be used interchangably. See the ROM code for the original declaration of struct ets_ds_data_t.

```
uint32_t iv[ESP_DS_IV_BIT_LEN / 32]
  IV value used to encrypt ‘c’
```

```
uint8_t c[ESP_DS_C_LEN]
  Encrypted Digital Signature parameters. Result of AES-CBC encryption of plaintext values. Includes an encrypted message digest.
```

```
struct esp_ds_p_data_t

Plaintext parameters used by Digital Signature.

This is only used for encrypting the RSA parameters by calling esp_ds_encrypt_params(). Afterwards, the result can be stored in flash or in other persistent memory. The encryption is a prerequisite step before any signature operation can be done.

Note: Y, M, Rb, & M_Prime must all be in little endian format.
```

Public Members

```
uint32_t Y[ESP_DS_SIGNATURE_MAX_BIT_LEN / 32]
  RSA exponent.
```

```
uint32_t M[ESP_DS_SIGNATURE_MAX_BIT_LEN / 32]
  RSA modulus.
```

```
uint32_t Rb[ESP_DS_SIGNATURE_MAX_BIT_LEN / 32]
  RSA r inverse operand.
```

```
uint32_t M_prime
  RSA M prime operand.
```

```
uint32_t length
  RSA length in words (32 bit)
```
Chapter 2. API Reference

Macros

ESP_DS_IV_BIT_LEN
ESP_DS_IV_LEN

ESP_DS_SIGNATURE_MAX_BIT_LEN
ESP_DS_SIGNATURE_MD_BIT_LEN
ESP_DS_SIGNATURE_M_PRIME_BIT_LEN
ESP_DS_SIGNATURE_L_BIT_LEN
ESP_DS_SIGNATURE_PADDING_BIT_LEN
ESP_DS_C_LEN

Type Definitions
typedef struct esp_ds_context esp_ds_context_t

typedef struct esp_digital_signature_data esp_ds_data_t

Encrypted private key data. Recommended to store in flash in this format.

Note: This struct has to match to one from the ROM code! This documentation is mostly taken from there.

Enumerations
enum esp_digital_signature_length_t

Values:

enumerator ESP_DS_RSA_1024
enumerator ESP_DS_RSA_2048
enumerator ESP_DS_RSA_3072
enumerator ESP_DS_RSA_4096

2.5.11 Inter-Integrated Circuit (I2C)

Introduction

I2C is a serial, synchronous, multi-device, half-duplex communication protocol that allows co-existence of multiple masters and slaves on the same bus. I2C uses two bidirectional open-drain lines: serial data line (SDA) and serial clock line (SCL), pulled up by resistors.

ESP32-S2 has 2 I2C controller (also called port), responsible for handling communication on the I2C bus. A single I2C controller can be a master or a slave.
Typically, an I2C slave device has a 7-bit address or 10-bit address. ESP32-S2 supports both I2C Standard-mode (Sm) and Fast-mode (Fm) which can go up to 100KHz and 400KHz respectively.

**Warning:** The clock frequency of SCL in master mode should not be larger than 400 KHz

**Note:** The frequency of SCL is influenced by both the pull-up resistor and the wire capacitance. Therefore, users are strongly recommended to choose appropriate pull-up resistors to make the frequency accurate. The recommended value for pull-up resistors usually ranges from 1K Ohms to 10K Ohms.

Keep in mind that the higher the frequency, the smaller the pull-up resistor should be (but not less than 1 KOHms). Indeed, large resistors will decline the current, which will increase the clock switching time and reduce the frequency. We usually recommend a range of 2 KOHms to 5 KOHms, but users may also need to make some adjustments depending on their current draw requirements.

### I2C Clock Configuration

- `i2c_clock_source_t::I2C_CLK_SRC_DEFAULT`: Default I2C source clock.
- `i2c_clock_source_t::I2C_CLK_SRC_APB`: APB clock as I2C clock source.
- `i2c_clock_source_t::I2C_CLK_SRC_REF_TICK`: 1MHZ clock.

### I2C File Structure

![I2C file structure diagram](image)

**Fig. 5: I2C file structure**

**Public headers that need to be included in the I2C application**

- `i2c.h`: The header file of legacy I2C APIs (for apps using legacy driver).
• `i2c_master.h`: The header file that provides standard communication mode specific APIs (for apps using new driver with master mode).
• `i2c_slave.h`: The header file that provides standard communication mode specific APIs (for apps using new driver with slave mode).

**Note:** The legacy driver cannot coexist with the new driver. Include `i2c.h` to use the legacy driver or the other two headers to use the new driver. Please keep in mind that the legacy driver is now deprecated and will be removed in future.

**Public headers that have been included in the headers above**

• `i2c_types_legacy.h`: The legacy public types that only used in the legacy driver.
• `i2c_types.h`: The header file that provides public types.

**Functional Overview**

The I2C driver offers following services:

• **Resource Allocation** - covers how to allocate I2C bus with properly set of configurations. It also covers how to recycle the resources when they finished working.
• **I2C Master Controller** - covers behavior of I2C master controller. Introduce data transmit, data receive, and data transmit and receive.
• **I2C Slave Controller** - covers behavior of I2C slave controller. Involve data transmit and data receive.
• **Power Management** - describes how different source clock will affect power consumption.
• **IRAM Safe** - describes tips on how to make the I2C interrupt work better along with a disabled cache.
• **Thread Safety** - lists which APIs are guaranteed to be thread safe by the driver.
• **Kconfig Options** - lists the supported Kconfig options that can bring different effects to the driver.

**Resource Allocation** Both I2C master bus and I2C slave bus, when supported, are represented by `i2c_bus_handle_t` in the driver. The available ports are managed in a resource pool that allocates a free port on request.

**Install I2C master bus and device** The I2C master is designed based on bus-device model. So `i2c_master_bus_config_t` and `i2c_device_config_t` are required separately to allocate the I2C master bus instance and I2C device instance.

I2C master bus requires the configuration that specified by `i2c_master_bus_config_t`:

• `i2c_master_bus_config_t::i2c_port` sets the I2C port used by the controller.
• `i2c_master_bus_config_t::sda_io_num` sets the GPIO number for the serial data bus (SDA).
• `i2c_master_bus_config_t::scl_io_num` sets the GPIO number for the serial clock bus (SCL).
• `i2c_master_bus_config_t::clk_source` selects the source clock for I2C bus. The available clocks are listed in `i2c_clock_source_t`. For the effect on power consumption of different clock source, please refer to **Power Management** section.
• `i2c_master_bus_config_t::glitch_ignore_cnt` sets the glitch period of master bus, if the glitch period on the line is less than this value, it can be filtered out, typically value is 7.
• `i2c_master_bus_config_t::intr_priority` Set the priority of the interrupt. If set to 0, then the driver will use a interrupt with low or medium priority (priority level may be one of 1, 2 or 3), otherwise use the priority indicated by `i2c_master_bus_config_t::intr_priority` Please use the number form (1,2,3) , not the bitmask form ((1<<1),(1<<2),(1<<3)).
• `i2c_master_bus_config_t::trans_queue_depth` Depth of internal transfer queue. Only valid in asynchronous transaction.
• `i2c_master_bus_config_t::enable_internal_pullup` Enable internal pullups. Note: This is not strong enough to pullup buses under high-speed frequency. A suitable external pullup is recommended.

If the configurations in `i2c_master_bus_config_t` is specified, users can call `i2c_new_master_bus()` to allocate and initialize an I2C master bus. This function will return an I2C bus handle if it runs correctly. Specifically, when there are no more I2C port available, this function will return ESP_ERR_NOT_FOUND error.
Chapter 2. API Reference

Fig. 6: I2C master bus-device module

I2C master device requires the configuration that specified by \texttt{i2c_device_config_t}:

- \texttt{i2c_device_config_t::dev_addr_length} configure the address bit length of the slave device. User can choose from enumerator I2C_ADDR_BIT_LEN_7 or I2C_ADDR_BIT_LEN_10 (if supported).
- \texttt{i2c_device_config_t::device_address} I2C device raw address. Please parse the device address to this member directly. For example, the device address is 0x28, then parse 0x28 to \texttt{i2c_device_config_t::device_address}, don’t carry a write/read bit.
- \texttt{i2c_device_config_t::scl_speed_hz} set the scl line frequency of this device.

Once the \texttt{i2c_device_config_t} structure is populated with mandatory parameters, users can call \texttt{i2c_master_bus_add_device()} to allocate an I2C device instance and mounted to the master bus then. This function will return an I2C device handle if it runs correctly. Specifically, when the I2C bus is not initialized properly, calling this function will result in a \texttt{ESP_ERR_INVALID_ARG} error.

```c
#include "driver/i2c_master.h"

i2c_master_bus_config_t i2c_mst_config = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
    .flags.enable_internal_pullup = true,
};

i2c_master_bus_handle_t bus_handle;
ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));

i2c_device_config_t dev_cfg = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = 0x58,
    .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg, &dev_handle));
```
**Uninstall I2C master bus and device** If a previously installed I2C bus or device is no longer needed, it’s recommended to recycle the resource by calling `i2c_master_bus_rm_device()` or `i2c_del_master_bus()`, so that to release the underlying hardware.

**Install I2C slave device** I2C slave requires the configuration that specified by `i2c_slave_config_t`:

- `i2c_slave_config_t::i2c_port` sets the I2C port used by the controller.
- `i2c_slave_config_t::sda_io_num` sets the GPIO number for serial data bus (SDA).
- `i2c_slave_config_t::scl_io_num` sets the GPIO number for serial clock bus (SCL).
- `i2c_slave_config_t::clk_source` selects the source clock for I2C bus. The available clocks are listed in `i2c_clock_source_t`. For the effect on power consumption of different clock source, please refer to Power Management section.
- `i2c_slave_config_t::send_buf_depth` sets the sending buffer length.
- `i2c_slave_config_t::slave_addr` sets the slave address.
- `i2c_master_bus_config_t::intr_priority` Set the priority of the interrupt. If set to 0, then the driver will use a interrupt with low or medium priority (priority level may be one of 1,2 or 3), otherwise use the priority indicated by `i2c_master_bus_config_t::intr_priority` Please use the number form (1, 2, 3), not the bitmask form ((1<<1),(1<<2),(1<<3)). Please pay attention that once the interrupt priority is set, it cannot be changed until `i2c_del_master_bus()` is called.
- `i2c_slave_config_t::addr_bit_len` sets true if you need the slave to have a 10-bit address.

Once the `i2c_slave_config_t` structure is populated with mandatory parameters, users can call `i2c_new_slave_device()` to allocate and initialize an I2C master bus. This function will return an I2C bus handle if it runs correctly. Specifically, when there are no more I2C port available, this function will return `ESP_ERR_NOT_FOUND` error.

```c
i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .send_buf_depth = 256,
    .scl_io_num = I2C_SLAVE_SCL_IO,
    .sda_io_num = I2C_SLAVE_SDA_IO,
    .slave_addr = 0x58,
};

i2c_slave_dev_handle_t slave_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &slave_handle));
```

**Uninstall I2C slave device** If a previously installed I2C bus is no longer needed, it’s recommended to recycle the resource by calling `i2c_del_slave_device()`, so that to release the underlying hardware.

**I2C Master Controller** After installing the i2c master driver by `i2c_new_master_bus()`, ESP32-S2 is ready to communicate with other I2C devices. I2C APIs allow the standard transactions. Like the wave as follows:

![Standard I2C Transaction Timing Diagram](image)

---

Submit Document Feedback
**I2C Master Write** After installing I2C master bus successfully, you can simply call `i2c_master_transmit()` to write data to the slave device. The principle of this function can be explained by following chart.

In order to organize the process, the driver uses a command link, that should be populated with a sequence of commands and then passed to I2C controller for execution.

![Fig. 7: I2C master write to slave](image)

Simple example for writing data to slave:

```c
#define DATA_LENGTH 100
i2c_master_bus_config_t i2c_mst_config = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = I2C_PORT_NUM_0,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
};
i2c_master_bus_handle_t bus_handle;
ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));
i2c_device_config_t dev_cfg = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = 0x58,
    .scl_speed_hz = 100000,
};
i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg, &dev_handle));
ESP_ERROR_CHECK(i2c_master_transmit(dev_handle, data_wr, DATA_LENGTH, -1));
```

**I2C Master Read** After installing I2C master bus successfully, you can simply call `i2c_master_receive()` to read data from the slave device. The principle of this function can be explained by following chart.

![Fig. 8: I2C master read from slave](image)

Simple example for reading data from slave:

```c
#define DATA_LENGTH 100
i2c_master_bus_config_t i2c_mst_config = {
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = I2C_PORT_NUM_0,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .glitch_ignore_cnt = 7,
};
i2c_master_bus_handle_t bus_handle;
```
ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));

i2c_device_config_t dev_cfg = {
  .dev_addr_length = I2C_ADDR_BIT_LEN_7,
  .device_address = 0x58,
  .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg, &dev_handle));

i2c_master_receive(dev_handle, data_rd, DATA_LENGTH, -1);

### I2C Master Write and Read

Some I2C device needs write configurations before reading data from it, therefore, an interface called `i2c_master_transmit_receive()` can help. The principle of this function can be explained by following chart.

Fig. 9: I2C master write to slave and read from slave

Simple example for writing and reading from slave:

```c
i2c_device_config_t dev_cfg = {
  .dev_addr_length = I2C_ADDR_BIT_LEN_7,
  .device_address = 0x58,
  .scl_speed_hz = 100000,
};

i2c_master_dev_handle_t dev_handle;
ESP_ERROR_CHECK(i2c_master_bus_add_device(I2C_PORT_NUM_0, &dev_cfg, &dev_handle));

uint8_t buf[20] = {0x20};
uint8_t buffer[2];
ESP_ERROR_CHECK(i2c_master_transmit_receive(i2c_bus_handle, buf, sizeof(buf), buffer, 2, -1));
```

### I2C Master Probe

I2C driver can use `i2c_master_probe()` to detect whether the specific device has been connected on I2C bus. If this function return ESP_OK, that means the device has been detected.

Fig. 10: I2C master probe

Simple example for probing an I2C device:

```c
i2c_master_bus_config_t i2c_mst_config_1 = {
  .clk_source = I2C_CLK_SRC_DEFAULT,
  .i2c_port = TEST_I2C_PORT,
  .scl_io_num = I2C_MASTER_SCL_IO,
  .sda_io_num = I2C_MASTER_SDA_IO,
  .glitch_ignore_cnt = 7,
  .flags.enable_internal_pullup = true,
};
```
Chapter 2. API Reference

(continued from previous page)

```c
i2c_master_bus_handle_t bus_handle;
ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config_1, &bus_handle));
ESP_ERROR_CHECK(i2c_master_probe(bus_handle, 0x22, -1));
ESP_ERROR_CHECK(i2c_del_master_bus(bus_handle));
```

**I2C Slave Controller**  After installing the i2c slave driver by `i2c_new_slave_device()`, ESP32-S2 is ready to communicate with other I2C master as a slave.

**I2C Slave Write**  The send buffer of the I2C slave is used as a FIFO to store the data to be sent. The data will queue up until the master requests them. You can call `i2c_slave_transmit()` to transfer data.

Simple example for writing data to FIFO:

```c
uint8_t *data_wr = (uint8_t *) malloc(DATA_LENGTH);

i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7, // 7-bit address
    .clk_source = I2C_CLK_SRC_DEFAULT, // set the clock source
    .i2c_port = 0, // set I2C port number
    .send_buf_depth = 256, // set tx buffer length
    .scl_io_num = 2, // SCL gpio number
    .sda_io_num = 1, // SDA gpio number
    .slave_addr = 0x58, // slave address
};

i2c_bus_handle_t i2c_bus_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &i2c_bus_handle));

for (int i = 0; i < DATA_LENGTH; i++) {
    data_wr[i] = i;
}

ESP_ERROR_CHECK(i2c_slave_transmit(i2c_bus_handle, data_wr, DATA_LENGTH, 10000));
```

**I2C Slave Read**  Whenever the master writes data to the slave, the slave will automatically store data in the receive buffer. This allows the slave application to call the function `i2c_slave_receive()` as its own discretion. As `i2c_slave_receive()` is designed as a non-blocking interface. So the user needs to register callback `i2c_slave_register_event_callbacks()` to know when the receive has finished.

```c
static IRAM_ATTR bool i2c_slave_rx_done_callback(i2c_slave_dev_handle_t channel,
        const i2c_slave_rx_done_event_data_t *edata, void *user_data)
{
    BaseType_t high_task_wakeup = pdFALSE;
    QueueHandle_t receive_queue = (QueueHandle_t)user_data;
    xQueueSendFromISR(receive_queue, edata, &high_task_wakeup);
    return high_task_wakeup == pdTRUE;
}

uint8_t *data_rd = (uint8_t *) malloc(DATA_LENGTH);
uint32_t size_rd = 0;

i2c_slave_config_t i2c_slv_config = {
    .addr_bit_len = I2C_ADDR_BIT_LEN_7,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .i2c_port = TEST_I2C_PORT,
    .send_buf_depth = 256,
    .scl_io_num = I2C_SLAVE_SCL_IO,
};
```

(continues on next page)
Register Event Callbacks

**I2C master callbacks** When an I2C master bus triggers an interrupt, a specific event will be generated and notify the CPU. If you have some functions that need to be called when those events occurred, you can hook your functions to the ISR (Interrupt Service Routine) by calling `i2c_master_register_event_callbacks()`. Since the registered callback functions are called in the interrupt context, user should ensure the callback function doesn’t attempt to block (e.g. by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The callback functions are required to return a boolean value, to tell the ISR whether a high priority task is woke up by it.

I2C master event callbacks are listed in the `i2c_master_event_callbacks_t`.

Although I2C is a synchronous communication protocol, we also support asynchronous behavior by registering above callback. In this way, I2C APIs will be non-blocking interface. But note that on the same bus, only one device can adopt asynchronous operation.

**Important:** I2C master asynchronous transaction is still an experimental feature. (The issue is when asynchronous transaction is very large, it will cause memory problem.)

- `i2c_master_event_callbacks_t::on_recv_done` sets a callback function for master “transaction-done” event. The function prototype is declared in `i2c_master_callback_t`.

**I2C slave callbacks** When an I2C slave bus triggers an interrupt, a specific event will be generated and notify the CPU. If you have some function that needs to be called when those events occurred, you can hook your function to the ISR (Interrupt Service Routine) by calling `i2c_slave_register_event_callbacks()`. Since the registered callback functions are called in the interrupt context, user should ensure the callback function doesn’t attempt to block (e.g. by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The callback function has a boolean return value, to tell the caller whether a high priority task is woke up by it.

I2C slave event callbacks are listed in the `i2c_slave_event_callbacks_t`.

- `i2c_slave_event_callbacks_t::on_recv_done` sets a callback function for “receive-done” event. The function prototype is declared in `i2c_slave_received_callback_t`.

```c
.sda_io_num = I2C_SLAVE_SDA_IO,
.slave_addr = 0x58,
};
i2c_slave_dev_handle_t slave_handle;
ESP_ERROR_CHECK(i2c_new_slave_device(&i2c_slv_config, &slave_handle));
s_receive_queue = xQueueCreate(1, sizeof(i2c_slave_rx_done_event_data_t));
i2c_slave_event_callbacks_t cbs = {
    .on_recv_done = i2c_slave_rx_done_callback,
};
ESP_ERROR_CHECK(i2c_slave_register_event_callbacks(slave_handle, &cbs, s_receive_);
rx_data;
ESP_ERROR_CHECK(i2c_slave_receive(slave_handle, data_rd, DATA_LENGTH));
xQueueReceive(s_receive_queue, &rx_data, pdMS_TO_TICKS(10000));
// Receive done.
```
**Power Management**  When the power management is enabled (i.e. `CONFIG_PM_ENABLE` is on), the system will adjust or stop the source clock of I2C fifo before going into light sleep, thus potentially changing the I2C signals and leading to transmitting or receiving invalid data.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type `ESP_PM_APB_FREQ_MAX`. Whenever user creates an I2C bus that has selected `I2C_CLK_SRC_APB` as the clock source, the driver will guarantee that the power management lock is acquired when I2C operations begin and release the lock automatically when I2C operations finish.

If the controller clock source is selected to `I2C_CLK_SRC_REF_TICK`, then the driver won’t install power management lock for it, which is more suitable for a low power application as long as the source clock can still provide sufficient resolution.

**IRAM Safe**  By default, the I2C interrupt will be deferred when the Cache is disabled for reasons like writing/erasing Flash. Thus the event callback functions will not get executed in time, which is not expected in a real-time application.

There’s a Kconfig option `CONFIG_I2C_ISR_IRAM_SAFE` that will:

1. Enable the interrupt being serviced even when cache is disabled
2. Place all functions that used by the ISR into IRAM
3. Place driver object into DRAM (in case it’s mapped to PSRAM by accident)

This will allow the interrupt to run while the cache is disabled but will come at the cost of increased IRAM consumption.

**Thread Safety**  The factory function `i2c_new_master_bus()` and `i2c_new_slave_device()` are guaranteed to be thread safe by the driver, which means, user can call them from different RTOS tasks without protection by extra locks. Other public I2C APIs are not thread safe. which means the user should avoid calling them from multiple tasks, if user strongly needs to call them in multiple tasks, please add extra lock.

**Kconfig Options**

- `CONFIG_I2C_ISR_IRAM_SAFE` controls whether the default ISR handler can work when cache is disabled, see also `IRAM Safe` for more information.
- `CONFIG_I2C_ENABLE_DEBUG_LOG` is used to enable the debug log at the cost of increased firmware binary size.

**API Reference**

**Header File**

- `components/driver/i2c/include/driver/i2c_master.h`

This header file can be included with:

```
#include "driver/i2c_master.h"
```

This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```
REQUIRES driver
```

or

```
PRIV_REQUIRES driver
```

**Functions**

```c
esp_err_t i2c_new_master_bus (const i2c_master_bus_config_t *bus_config, i2c_master_bus_handle_t *ret_bus_handle)
```

Allocate an I2C master bus.
**Parameters**

- **bus_config** [in] I2C master bus configuration.
- **ret_bus_handle** [out] I2C bus handle

**Returns**

- ESP_OK: I2C master bus initialized successfully.
- ESP_ERR_INVALID_ARG: I2C bus initialization failed because of invalid argument.
- ESP_ERR_NO_MEM: Create I2C bus failed because of out of memory.
- ESP_ERR_NOT_FOUND: No more free bus.

```c
esp_err_t i2c_master_bus_add_device(i2c_master_bus_handle_t bus_handle, const i2c_device_config_t *dev_config, i2c_master_dev_handle_t *ret_handle)
```

Add I2C master BUS device.

**Parameters**

- **bus_handle** [in] I2C bus handle.
- **dev_config** [in] device config.
- **ret_handle** [out] device handle.

**Returns**

- ESP_OK: Create I2C master device successfully.
- ESP_ERR_INVALID_ARG: I2C bus initialization failed because of invalid argument.
- ESP_ERR_NO_MEM: Create I2C bus failed because of out of memory.

```c
esp_err_t i2c_del_master_bus(i2c_master_bus_handle_t bus_handle)
```

Deinitialize the I2C master bus and delete the handle.

**Parameters**

- **bus_handle** [in] I2C bus handle.

**Returns**

- ESP_OK: Delete I2C bus success, otherwise, failed.
- Otherwise: Some module delete failed.

```c
esp_err_t i2c_master_bus_rm_device(i2c_master_dev_handle_t handle)
```

I2C master bus delete device.

**Parameters**

- **handle** – i2c device handle

**Returns**

- ESP_OK: If device is successfully deleted.

```c
esp_err_t i2c_master_transmit(i2c_master_dev_handle_t i2c_dev, const uint8_t *write_buffer, size_t write_size, int xfer_timeout_ms)
```

Perform a write transaction on the I2C bus. The transaction will be undergoing until it finishes or it reaches the timeout provided.

**Note:** If a callback was registered with `i2c_master_register_event_callbacks`, the transaction will be asynchronous, and thus, this function will return directly, without blocking. You will get finish information from callback. Besides, data buffer should always be completely prepared when callback is registered, otherwise, the data will get corrupt.

**Parameters**

- **i2c_dev** – [in] I2C master device handle that created by `i2c_master_bus_add_device`.
- **write_buffer** [in] Data bytes to send on the I2C bus.
- **write_size** [in] Size, in bytes, of the write buffer.

**Returns**

- ESP_OK: I2C master transmit success
- ESP_ERR_INVALID_ARG: I2C master transmit parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.
**esp_err_t i2c_master_transmit_receive**(i2c_master_dev_handle_t *i2c_dev, const uint8_t *write_buffer, size_t write_size, uint8_t *read_buffer, size_t read_size, int xfer_timeout_ms)

Perform a write-read transaction on the I2C bus. The transaction will be undergoing until it finishes or it reaches the timeout provided.

**Note:** If a callback was registered with i2c_master_register_event_callbacks, the transaction will be asynchronous, and thus, this function will return directly, without blocking. You will get finish information from callback. Besides, data buffer should always be completely prepared when callback is registered, otherwise, the data will get corrupt.

**Parameters**

- **i2c_dev**  
  I2C master device handle that created by i2c_master_bus_add_device.
- **write_buffer**  
  Data bytes to send on the I2C bus.
- **write_size**  
  Size, in bytes, of the write buffer.
- **read_buffer**  
  Data bytes received from i2c bus.
- **read_size**  
  Size, in bytes, of the read buffer.
- **xfer_timeout_ms**  
  Wait timeout, in ms. Note: -1 means wait forever.

**Returns**

- ESP_OK: I2C master transmit-receive success
- ESP_ERR_INVALID_ARG: I2C master transmit parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.

**esp_err_t i2c_master_receive**(i2c_master_dev_handle_t *i2c_dev, uint8_t *read_buffer, size_t read_size, int xfer_timeout_ms)

Perform a read transaction on the I2C bus. The transaction will be undergoing until it finishes or it reaches the timeout provided.

**Note:** If a callback was registered with i2c_master_register_event_callbacks, the transaction will be asynchronous, and thus, this function will return directly, without blocking. You will get finish information from callback. Besides, data buffer should always be completely prepared when callback is registered, otherwise, the data will get corrupt.

**Parameters**

- **i2c_dev**  
  I2C master device handle that created by i2c_master_bus_add_device.
- **read_buffer**  
  Data bytes received from i2c bus.
- **read_size**  
  Size, in bytes, of the read buffer.
- **xfer_timeout_ms**  
  Wait timeout, in ms. Note: -1 means wait forever.

**Returns**

- ESP_OK: I2C master receive success
- ESP_ERR_INVALID_ARG: I2C master receive parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the bus is busy or hardware crash.

**esp_err_t i2c_master_probe**(i2c_master_bus_handle_t *bus_handle, uint16_t address, int xfer_timeout_ms)

Probe I2C address, if address is correct and ACK is received, this function will return ESP_OK.

**Parameters**

- **bus_handle**  
  I2C master device handle that created by i2c_master_bus_add_device.
- **address**  
  I2C device address that you want to probe.
Chapter 2. API Reference

- **xfer_timeout_ms** - [in] Wait timeout, in ms. Note: -1 means wait forever (Not recommended in this function).

**Returns**
- ESP_OK: I2C device probe successfully
- ESP_ERR_NOT_FOUND: I2C probe failed, doesn’t find the device with specific address you gave.
- ESP_ERR_TIMEOUT: Operation timeout (larger than xfer_timeout_ms) because the bus is busy or hardware crash.

```c
esp_err_t i2c_master_register_event_callbacks(i2c_master_dev_handle_t i2c_dev, const i2c_master_event_callbacks_t *cbs, void *user_data)
```

Register I2C transaction callbacks for a master device.

**Note:** User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

**Note:** When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

**Note:** If the callback is used for helping asynchronous transaction. On the same bus, only one device can be used for performing asynchronous operation.

**Parameters**
- `i2c_dev` - [in] I2C master device handle that created by `i2c_master_bus_add_device`
- `cbs` - [in] Group of callback functions
- `user_data` - [in] User data, which will be passed to callback functions directly

**Returns**
- ESP_OK: Set I2C transaction callbacks successfully
- ESP_ERR_INVALID_ARG: Set I2C transaction callbacks failed because of invalid argument
- ESP_FAIL: Set I2C transaction callbacks failed because of other error

```c
esp_err_t i2c_master_bus_reset(i2c_master_bus_handle_t bus_handle)
```

Reset the I2C master bus.

**Parameters**
- `bus_handle` - [in] I2C bus handle.

**Returns**
- ESP_OK: Reset succeed.
- ESP_ERR_INVALID_ARG: I2C master bus handle is not initialized.
- Otherwise: Reset failed.

```c
esp_err_t i2c_master_bus_wait_all_done(i2c_master_bus_handle_t bus_handle, int timeout_ms)
```

Wait for all pending I2C transactions done.

**Parameters**
- `bus_handle` - [in] I2C bus handle

**Returns**
- ESP_OK: Flush transactions successfully
- ESP_ERR_INVALID_ARG: Flush transactions failed because of invalid argument
- ESP_ERR_TIMEOUT: Flush transactions failed because of timeout
- ESP_FAIL: Flush transactions failed because of other error
Structures

struct i2c_master_bus_config_t
I2C master bus specific configurations.

Public Members

i2c_port_num_t i2c_port
I2C port number, -1 for auto selecting

gpio_num_t sda_io_num
GPIO number of I2C SDA signal, pulled-up internally

gpio_num_t scl_io_num
GPIO number of I2C SCL signal, pulled-up internally

i2c_clock_source_t clk_source
Clock source of I2C master bus, channels in the same group must use the same clock source

uint8_t glitch_ignore_cnt
If the glitch period on the line is less than this value, it can be filtered out, typically value is 7 (unit: I2C module clock cycle)

int intr_priority
I2C interrupt priority, if set to 0, driver will select the default priority (1,2,3).

size_t trans_queue_depth
Depth of internal transfer queue, increase this value can support more transfers pending in the background, only valid in asynchronous transaction. (Typically max_device_num * per_transaction)

uint32_t enable_internal_pullup
Enable internal pullups. Note: This is not strong enough to pullup buses under high-speed frequency. Recommend proper external pull-up if possible

struct i2c_master_bus_config_t::[anonymous] flags
I2C master config flags

struct i2c_device_config_t
I2C device configuration.

Public Members

i2c_addr_bit_len_t dev_addr_length
Select the address length of the slave device.

uint16_t device_address
I2C device raw address. (The 7/10 bit address without read/write bit)
uint32_t scl_speed_hz
I2C SCL line frequency.

struct i2c_master_event_callbacks_t
Group of I2C master callbacks, can be used to get status during transaction or doing other small things. But take care potential concurrency issues.

Note: The callbacks are all running under ISR context

Note: When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

i2c_master_callback_t on_trans_done
I2C master transaction finish callback

Header File

- components/driver/i2c/include/driver/i2c_slave.h
- This header file can be included with:

```c
#include "driver/i2c_slave.h"
```

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```cmake
REQUIRES driver
```

or

```cmake
PRIV_REQUIRES driver
```

Functions

esp_err_t i2c_new_slave_device (const i2c_slave_config_t *slave_config, i2c_slave_dev_handle_t *ret_handle)
Initialize an I2C slave device.

Parameters

- slave_config [in] I2C slave device configurations
- ret_handle [out] Return a generic I2C device handle

Returns

- ESP_OK: I2C slave device initialized successfully
- ESP_ERR_INVALID_ARG: I2C device initialization failed because of invalid argument.
- ESP_ERR_NO_MEM: Create I2C device failed because of out of memory.

esp_err_t i2c_del_slave_device (i2c_slave_dev_handle_t i2c_slave)
Deinitialize the I2C slave device.

Parameters i2c_slave [in] I2C slave device handle that created by i2c_new_slave_device.

Returns

- ESP_OK: Delete I2C device successfully.
- ESP_ERR_INVALID_ARG: I2C device initialization failed because of invalid argument.
esp_err_t i2c_slave_receive (i2c_slave_dev_handle_t i2c_slave, uint8_t *data, size_t buffer_size)

Read bytes from I2C internal buffer. Start a job to receive I2C data.

**Note:** This function is non-blocking, it initiates a new receive job and then returns. User should check the received data from the on_recv_done callback that registered by i2c_slave_register_event_callbacks().

**Parameters**

- **i2c_slave** - [in] I2C slave device handle that created by i2c_new_slave_device.
- **data** - [out] Buffer to store data from I2C fifo. Should be valid until on_recv_done is triggered.
- **buffer_size** - [in] Buffer size of data that provided by users.

**Returns**

- ESP_OK: I2C slave receive success.
- ESP_ERR_INVALID_ARG: I2C slave receive parameter invalid.
- ESP_ERR_NOT_SUPPORTED: This function should be work in fifo mode, but I2C_SLAVE_NONFIFO mode is configured

esp_err_t i2c_slave_transmit (i2c_slave_dev_handle_t i2c_slave, const uint8_t *data, int size, int xfer_timeout_ms)

Write bytes to internal ringbuffer of the I2C slave data. When the TX fifo empty, the ISR will fill the hardware FIFO with the internal ringbuffer’s data.

**Note:** If you connect this slave device to some master device, the data transaction direction is from slave device to master device.

**Parameters**

- **i2c_slave** - [in] I2C slave device handle that created by i2c_new_slave_device.
- **data** - [in] Buffer to write to slave fifo, can pickup by master. Can be freed after this function returns. Equal or larger than size.
- **size** - [in] In bytes, of data buffer.

**Returns**

- ESP_OK: I2C slave transmit success.
- ESP_ERR_INVALID_ARG: I2C slave transmit parameter invalid.
- ESP_ERR_TIMEOUT: Operation timeout(larger than xfer_timeout_ms) because the device is busy or hardware crash.
- ESP_ERR_NOT_SUPPORTED: This function should be work in fifo mode, but I2C_SLAVE_NONFIFO mode is configured

esp_err_t i2c_slave_register_event_callbacks (i2c_slave_dev_handle_t i2c_slave, const i2c_slave_event_callbacks_t *cbs, void *user_data)

Set I2C slave event callbacks for I2C slave channel.

**Note:** User can deregister a previously registered callback by calling this function and setting the callback member in the cbs structure to NULL.

**Note:** When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The user_data
should also reside in SRAM.

Parameters
- `i2c_slave` — [in] I2C slave device handle that created by `i2c_new_slave_device`.
- `cbs` — [in] Group of callback functions
- `user_data` — [in] User data, which will be passed to callback functions directly

Returns
- `ESP_OK`: Set I2C transaction callbacks successfully
- `ESP_ERR_INVALID_ARG`: Set I2C transaction callbacks failed because of invalid argument
- `ESP_FAIL`: Set I2C transaction callbacks failed because of other error

Structures

```c
struct i2c_slave_config_t
```

I2C slave specific configurations.

Public Members

```c
i2c_port_num_t i2c_port
```

I2C port number, -1 for auto selecting

```c
gpio_num_t sda_io_num
```

SDA IO number used by I2C bus

```c
gpio_num_t scl_io_num
```

SCL IO number used by I2C bus

```c
i2c_clock_source_t clk_source
```

Clock source of I2C bus.

```c
uint32_t send_buf_depth
```

Depth of internal transfer ringbuffer, increase this value can support more transfers pending in the background

```c
uint16_t slave_addr
```

I2C slave address

```c
i2c_addr_bit_len_t addr_bit_len
```

I2C slave address in bit length

```c
int intr_priority
```

I2C interrupt priority, if set to 0, driver will select the default priority (1,2,3).

```c
struct i2c_slave_config_t::[anonymous] flags
```

I2C slave config flags
struct i2c_slave_event_callbacks_t
  Group of I2C slave callbacks (e.g. get i2c slave stretch cause). But take care of potential concurrency issues.

  **Note:** The callbacks are all running under ISR context

  **Note:** When CONFIG_I2C_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

**Public Members**

*i2c_slave_received_callback_t on_recv_done*
  I2C slave receive done callback

**Header File**

- components/driver/i2c/include/driver/i2c_types.h
- This header file can be included with:
  ```
  #include "driver/i2c_types.h"
  ```

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
  ```
  REQUIRES driver
  ```
  or
  ```
  PRIV_REQUIRES driver
  ```

**Structures**

struct i2c_master_event_data_t
  Data type used in I2C event callback.

**Public Members**

*i2c_master_event_t event*
  The I2C hardware event that I2C callback is called.

struct i2c_slave_rx_done_event_data_t
  Event structure used in I2C slave.

**Public Members**

uint8_t *buffer
  Pointer for buffer received in callback.
Type Definitions

typedef int i2c_port_num_t
I2C port number.

typedef struct i2c_master_bus_t *i2c_master_bus_handle_t
Type of I2C master bus handle.

typedef struct i2c_master_dev_t *i2c_master_dev_handle_t
Type of I2C master bus device handle.

typedef struct i2c_slave_dev_t *i2c_slave_dev_handle_t
Type of I2C slave device handle.

typedef bool (*i2c_master_callback_t)(i2c_master_dev_handle_t i2c_dev, const i2c_master_event_data_t *evt_data, void *arg)
An callback for I2C transaction.

Param i2c_dev [in] Handle for I2C device.
Param evt_data [out] I2C capture event data, fed by driver
Param arg [in] User data, set in i2c_master_register_event_callbacks()
Return Whether a high priority task has been waken up by this function

typedef bool (*i2c_slave_received_callback_t)(i2c_slave_dev_handle_t i2c_slave, const i2c_slave_rx_done_event_data_t *evt_data, void *arg)
Callback signature for I2C slave.

Param i2c_slave [in] Handle for I2C slave.
Param evt_data [out] I2C capture event data, fed by driver
Param user_ctx [in] User data, set in i2c_slave_register_event_callbacks()
Return Whether a high priority task has been waken up by this function

Enumerations

enum i2c_master_status_t
Enumeration for I2C fsm status.

Values:

enumerator I2C_STATUS_READ
read status for current master command

enumerator I2C_STATUS_WRITE
write status for current master command

enumerator I2C_STATUS_START
Start status for current master command

enumerator I2C_STATUS_STOP
stop status for current master command

enumerator I2C_STATUS_IDLE
idle status for current master command
enumerator **I2C_STATUS_ACK_ERROR**  
ack error status for current master command

enumerator **I2C_STATUS_DONE**  
I2C command done

enumerator **I2C_STATUS_TIMEOUT**  
I2C bus status error, and operation timeout

enum **i2c_master_event_t**  
*Values:*

enumerator **I2C_EVENT_ALIVE**  
i2c bus in alive status.

enumerator **I2C_EVENT_DONE**  
i2c bus transaction done

enumerator **I2C_EVENT_NACK**  
i2c bus nack

**Header File**

- components/hal/include/hal/i2c_types.h
- This header file can be included with:

  ```
  #include "hal/i2c_types.h"
  ```

**Structures**

struct **i2c_hal_clk_config_t**  
Data structure for calculating I2C bus timing.

**Public Members**

- `uint16_t clkm_div`  
  I2C core clock devider

- `uint16_t scl_low`  
  I2C scl low period

- `uint16_t scl_high`  
  I2C scl high period

- `uint16_t scl_wait_high`  
  I2C scl wait_high period

- `uint16_t sda_hold`  
  I2C scl low period
uint16_t sda_sample
I2C sda sample time

uint16_t setup
I2C start and stop condition setup period

uint16_t hold
I2C start and stop condition hold period

uint16_t tout
I2C bus timeout period

**Type Definitions**

typedef soc_periph_i2c_clk_src_t i2c_clock_source_t
I2C group clock source.

**Enumerations**

denum i2c_port_t
I2C port number, can be I2C_NUM_0 ~ (I2C_NUM_MAX-1).

*Values:*

enumerator I2C_NUM_0
I2C port 0

enumerator I2C_NUM_1
I2C port 1

enumerator I2C_NUM_MAX
I2C port max

enum i2c_addr_bit_len_t
Enumeration for I2C device address bit length.

*Values:*

enumerator I2C_ADDR_BIT_LEN_7
i2c address bit length 7

enum i2c_mode_t

*Values:*

enumerator I2C_MODE_SLAVE
I2C slave mode

enumerator I2C_MODE_MASTER
I2C master mode
Chapter 2. API Reference

enumerator **I2C_MODE_MAX**

class **i2c_rw_t**

    **Values:**

    enumerator **I2C_MASTER_WRITE**
    12C write data

    enumerator **I2C_MASTER_READ**
    12C read data

enum **i2c_trans_mode_t**

    **Values:**

    enumerator **I2C_DATA_MODE_MSB_FIRST**
    12C data msb first

    enumerator **I2C_DATA_MODE_LSB_FIRST**
    12C data lsb first

    enumerator **I2C_DATA_MODE_MAX**

enum **i2c_addr_mode_t**

    **Values:**

    enumerator **I2C_ADDR_BIT_7**
    12C 7bit address for slave mode

    enumerator **I2C_ADDR_BIT_10**
    12C 10bit address for slave mode

    enumerator **I2C_ADDR_BIT_MAX**

enum **i2c_ack_type_t**

    **Values:**

    enumerator **I2C_MASTER_ACK**
    12C ack for each byte read

    enumerator **I2C_MASTER_NACK**
    12C nack for each byte read

    enumerator **I2C_MASTER_LAST_NACK**
    12C nack for the last byte

    enumerator **I2C_MASTER_ACK_MAX**
Chapter 2. API Reference

enum i2c_slave_stretch_cause_t

Enum for I2C slave stretch causes.

Values:

enumerator I2C_SLAVE_STRETCH_CAUSE_ADDRESS_MATCH
    Stretching SCL low when the slave is read by the master and the address just matched

enumerator I2C_SLAVE_STRETCH_CAUSE_TX_EMPTY
    Stretching SCL low when TX FIFO is empty in slave mode

enumerator I2C_SLAVE_STRETCH_CAUSE_RX_FULL
    Stretching SCL low when RX FIFO is full in slave mode

enumerator I2C_SLAVE_STRETCH_CAUSE_SENDING_ACK
    Stretching SCL low when slave sending ACK

2.5.12 Inter-IC Sound (I2S)

Introduction

I2S (Inter-IC Sound) is a synchronous serial communication protocol usually used for transmitting audio data between two digital audio devices.

ESP32-S2 contains one I2S peripheral(s). These peripherals can be configured to input and output sample data via the I2S driver.

An I2S bus that communicates in standard or TDM mode consists of the following lines:

- **MCLK**: Master clock line. It is an optional signal depending on the slave side, mainly used for offering a reference clock to the I2S slave device.
- **BCLK**: Bit clock line. The bit clock for data line.
- **WS**: Word (Slot) select line. It is usually used to identify the vocal tract except PDM mode.
- **DIN/DOUT**: Serial data input/output line. Data will loopback internally if DIN and DOUT are set to a same GPIO.

Each I2S controller has the following features that can be configured by the I2S driver:

- Operation as system master or slave
- Capable of acting as transmitter or receiver
- DMA controller that allows stream sampling of data without requiring the CPU to copy each data sample

Each controller supports single RX or TX simplex communication. As RX and TX channels share a clock, they can only be combined with the same configuration to establish a full-duplex communication.

I2S File Structure

Public headers that need to be included in the I2S application are as follows:

- **i2s.h**: The header file that provides legacy I2S APIs (for apps using legacy driver).
- **i2s_std.h**: The header file that provides standard communication mode specific APIs (for apps using new driver with standard mode).
- **i2s_pdm.h**: The header file that provides PDM communication mode specific APIs (for apps using new driver with PDM mode).
• `i2s_tdm.h`: The header file that provides TDM communication mode specific APIs (for apps using new driver with TDM mode).

**Note:** The legacy driver cannot coexist with the new driver. Include `i2s.h` to use the legacy driver, or include the other three headers to use the new driver. The legacy driver might be removed in future.

**Public headers that have been included in the headers above are as follows:**

• `i2s_types_legacy.h`: The header file that provides legacy public types that are only used in the legacy driver.

• `i2s_types.h`: The header file that provides public types.

• `i2s_common.h`: The header file that provides common APIs for all communication modes.

**I2S Clock**

**Clock Source**

• `i2s_clock_src_t::I2S_CLK_SRC_DEFAULT`: Default PLL clock.

• `i2s_clock_src_t::I2S_CLK_SRC_PLL_160M`: 160 MHz PLL clock.

• `i2s_clock_src_t::I2S_CLK_SRC_APLL`: Audio PLL clock, which is more precise than `I2S_CLK_SRC_PLL_160M` in high sample rate applications. Its frequency is configurable according to the sample rate. However, if APLL has been occupied by EMAC or other channels, the APLL frequency cannot be changed, and the driver will try to work under this APLL frequency. If this frequency cannot meet the requirements of I2S, the clock configuration will fail.

**Clock Terminology**

• **Sample rate**: The number of sampled data in one second per slot.

• **SCLK**: Source clock frequency. It is the frequency of the clock source.

• **MCLK**: Master clock frequency. BCLK is generated from this clock. The MCLK signal usually serves as a reference clock and is mostly needed to synchronize BCLK and WS between I2S master and slave roles.
• **BCLK**: Bit clock frequency. Every tick of this clock stands for one data bit on data pin. The slot bit width configured in `i2s_std_slot_config_t::slot_bit_width` is equal to the number of BCLK ticks, which means there will be 8/16/24/32 BCLK ticks in one slot.

• **LRCK / WS**: Left/right clock or word select clock. For non-PDM mode, its frequency is equal to the sample rate.

**Note:** Normally, MCLK should be the multiple of sample rate and BCLK at the same time. The field `i2s_std_clk_config_t::mclk_multiple` indicates the multiple of MCLK to the sample rate. In most cases, `I2S_MCLK_MULTIPLE_256` should be enough. However, if `slot_bit_width` is set to `I2S_SLOT_BIT_WIDTH_24BIT`, to keep MCLK a multiple to the BCLK, `i2s_std_clk_config_t::mclk_multiple` should be set to multiples that are divisible by 3 such as `I2S_MCLK_MULTIPLE_384`. Otherwise, WS will be inaccurate.

### I2S Communication Mode

#### Overview of All Modes

<table>
<thead>
<tr>
<th>Target</th>
<th>Standard</th>
<th>PDM TX</th>
<th>PDM RX</th>
<th>TDM</th>
<th>ADC/DAC</th>
<th>LCD/Camera</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP32</td>
<td>I2S 0/1</td>
<td>I2S 0</td>
<td>I2S 0</td>
<td>none</td>
<td>I2S 0</td>
<td>I2S 0</td>
</tr>
<tr>
<td>ESP32-S2</td>
<td>I2S 0</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>I2S 0</td>
</tr>
<tr>
<td>ESP32-C3</td>
<td>I2S 0</td>
<td>I2S 0</td>
<td>none</td>
<td>I2S 0</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>ESP32-C6</td>
<td>I2S 0</td>
<td>I2S 0</td>
<td>none</td>
<td>I2S 0</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>ESP32-S3</td>
<td>I2S 0/1</td>
<td>I2S 0</td>
<td>I2S 0</td>
<td>I2S 0/1</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>ESP32-H2</td>
<td>I2S 0</td>
<td>I2S 0</td>
<td>none</td>
<td>I2S 0</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>ESP32-P4</td>
<td>I2S 0~2</td>
<td>I2S 0</td>
<td>I2S 0</td>
<td>I2S 0~2</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

**Standard Mode** In standard mode, there are always two sound channels, i.e., the left and right channels, which are called “slots”. These slots support 8/16/24/32-bit width sample data. The communication format for the slots mainly includes the followings:

- **Philips Format**: Data signal has one-bit shift comparing to the WS signal, and the duty of WS signal is 50%.

![Standard Philips Timing Diagram](image)

- **MSB Format**: Basically the same as Philips format, but without data shift.
**PCM Short Format**: Data has one-bit shift and meanwhile the WS signal becomes a pulse lasting for one BCLK cycle.

**LCD/Camera Mode**  LCD/Camera mode is only supported on I2S0 over a parallel bus. For LCD mode, I2S0 should work at master TX mode. For camera mode, I2S0 should work at slave RX mode. These two modes are not implemented by the I2S driver. Please refer to LCD for details about the LCD implementation. For more information, see ESP32-S2 Technical Reference Manual > I2S Controller (I2S) > LCD Mode [PDF].

**Functional Overview**

The I2S driver offers the following services:

**Resource Management**  There are three levels of resources in the I2S driver:

- **platform level**: Resources of all I2S controllers in the current target.
- **controller level**: Resources in one I2S controller.
- **channel level**: Resources of TX or RX channel in one I2S controller.

The public APIs are all channel-level APIs. The channel handle `i2s_chan_handle_t` can help users to manage the resources under a specific channel without considering the other two levels. The other two upper levels’ resources are private and are managed by the driver automatically. Users can call `i2s_new_channel()` to allocate a channel handle and call `i2s_del_channel()` to delete it.

**Power Management**  When the power management is enabled (i.e., `CONFIG_PM_ENABLE` is on), the system will adjust or stop the source clock of I2S before entering Light-sleep, thus potentially changing the I2S signals and leading to transmitting or receiving invalid data.

The I2S driver can prevent the system from changing or stopping the source clock by acquiring a power management lock. When the source clock is generated from APB, the lock type will be set to `esp_pm_lock_type_t::ESP_PM_APB_FREQ_MAX` and when the source clock is APLL (if supported), it
will be set to `esp_pm_lock_type_t::ESP_PM_NO_LIGHT_SLEEP`. Whenever the user is reading or writing via I2S (i.e., calling `i2s_channel_read()` or `i2s_channel_write()`), the driver guarantees that the power management lock is acquired. Likewise, the driver releases the lock after the reading or writing finishes.

**Finite State Machine**  There are three states for an I2S channel, namely, registered, ready, and running. Their relationship is shown in the following diagram:

![I2S Finite State Machine Diagram](image)

The `<mode>` in the diagram can be replaced by corresponding I2S communication modes, e.g., `std` for standard two-slot mode. For more information about communication modes, please refer to the *I2S Communication Mode* section.

**Data Transport**  The data transport of the I2S peripheral, including sending and receiving, is realized by DMA. Before transporting data, please call `i2s_channel_enable()` to enable the specific channel. When the sent or received data reaches the size of one DMA buffer, the `I2S_OUT_EOF` or `I2S_IN_SUC_EOF` interrupt will be triggered. Note that the DMA buffer size is not equal to `i2s_chan_config_t::dma_frame_num`. One frame here refers to all the sampled data in one WS circle. Therefore, `dma_buffer_size = dma_frame_num * slot_num * slot_bit_width / 8`. For the data transmitting, users can input the data by calling `i2s_channel_write()`. This function helps users to copy the data from the source buffer to the DMA TX buffer and wait for the transmission to finish. Then it will repeat until the sent bytes reach the given size. For the data receiving, the function `i2s_channel_read()` waits to receive the message queue which contains the DMA buffer address. It helps users copy the data from the DMA RX buffer to the destination buffer.

Both `i2s_channel_write()` and `i2s_channel_read()` are blocking functions. They keeps waiting until the whole source buffer is sent or the whole destination buffer is loaded, unless they exceed the max blocking time, where the error code `ESP_ERR_TIMEOUT` returns. To send or receive data asynchronously, callbacks can be registered by `i2s_channel_register_event_callback()`. Users are able to access the DMA buffer directly
in the callback function instead of transmitting or receiving by the two blocking functions. However, please be aware that it is an interrupt callback, so do not add complex logic, run floating operation, or call non-reentrant functions in the callback.

**Configuration** Users can initialize a channel by calling corresponding functions (i.e., `i2s_channel_init_std_mode()`, `i2s_channel_init_pdm_rx_mode()`, `i2s_channel_init_pdm_tx_mode()`, or `i2s_channel_init_tdm_mode()`) to a specific mode. If the configurations need to be updated after initialization, users have to first call `i2s_channel_disable()` to ensure that the channel has stopped, and then call corresponding reconfig functions, like `i2s_channel_reconfig_std_slot()`, `i2s_channel_reconfig_std_clock()`, and `i2s_channel_reconfig_std_gpio()`.

**IRAM Safe** By default, the I2S interrupt will be deferred when the cache is disabled for reasons like writing/erasing flash. Thus, the EOF interrupt will not get executed in time.

To avoid such case in real-time applications, you can enable the Kconfig option `CONFIG_I2S_ISR_IRAM_SAFE` that:

1. Keeps the interrupt being serviced even when the cache is disabled.
2. Places driver object into DRAM (in case it is linked to PSRAM by accident).

This allows the interrupt to run while the cache is disabled, but comes at the cost of increased IRAM consumption.

**Thread Safety** All the public I2S APIs are guaranteed to be thread safe by the driver, which means users can call them from different RTOS tasks without protection by extra locks. Notice that the I2S driver uses mutex lock to ensure the thread safety, thus these APIs are not allowed to be used in ISR.

**Kconfig Options**

- `CONFIG_I2S_ISR_IRAM_SAFE` controls whether the default ISR handler can work when the cache is disabled. See IRAM Safe for more information.
- `CONFIG_I2S_SUPPRESS_DEPRECATE_WARN` controls whether to suppress the compiling warning message while using the legacy I2S driver.
- `CONFIG_I2S_ENABLE_DEBUG_LOG` is used to enable the debug log output. Enable this option increases the firmware binary size.

**Application Example**

The examples of the I2S driver can be found in the directory `peripherals/i2s`. Here are some simple usages of each mode:

**Standard TX/RX Usage** Different slot communication formats can be generated by the following helper macros for standard mode. As described above, there are three formats in standard mode, and their helper macros are:

- `I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG`
- `I2S_STD_PCM_SLOT_DEFAULT_CONFIG`
- `I2S_STD_MSB_SLOT_DEFAULT_CONFIG`

The clock config helper macro is:

- `I2S_STD_CLK_DEFAULT_CONFIG`

Please refer to Standard Mode for information about STD API. And for more details, please refer to `driver/i2s/include/driver/i2s_std.h`. 
**STD TX Mode**  Take 16-bit data width for example. When the data in a `uint16_t` writing buffer are:

<table>
<thead>
<tr>
<th>data 0</th>
<th>data 1</th>
<th>data 2</th>
<th>data 3</th>
<th>data 4</th>
<th>data 5</th>
<th>data 6</th>
<th>data 7</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0x0002</td>
<td>0x0003</td>
<td>0x0004</td>
<td>0x0005</td>
<td>0x0006</td>
<td>0x0007</td>
<td>0x0008</td>
<td>…</td>
</tr>
</tbody>
</table>

Here is the table of the real data on the line with different `i2s_std_slot_config_t::slot_mode` and `i2s_std_slot_config_t::slot_mask`.

<table>
<thead>
<tr>
<th>data bit width</th>
<th>slot mode</th>
<th>slot mask</th>
<th>WS low</th>
<th>WS high</th>
<th>WS low</th>
<th>WS high</th>
<th>WS low</th>
<th>WS high</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>mono</td>
<td>left</td>
<td>0x0001</td>
<td>0x0000</td>
<td>0x0002</td>
<td>0x0000</td>
<td>0x0003</td>
<td>0x0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>right</td>
<td>0x0000</td>
<td>0x0001</td>
<td>0x0000</td>
<td>0x0002</td>
<td>0x0000</td>
<td>0x0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>both</td>
<td>0x0001</td>
<td>0x0001</td>
<td>0x0002</td>
<td>0x0002</td>
<td>0x0003</td>
<td>0x0003</td>
</tr>
<tr>
<td></td>
<td>stereo</td>
<td>left</td>
<td>0x0001</td>
<td>0x0001</td>
<td>0x0002</td>
<td>0x0000</td>
<td>0x0003</td>
<td>0x0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>right</td>
<td>0x0002</td>
<td>0x0000</td>
<td>0x0004</td>
<td>0x0000</td>
<td>0x0005</td>
<td>0x0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>both</td>
<td>0x0001</td>
<td>0x0002</td>
<td>0x0003</td>
<td>0x0004</td>
<td>0x0005</td>
<td>0x0006</td>
</tr>
</tbody>
</table>

**Note:** Similar for 8-bit and 32-bit data widths, the type of the buffer is better to be `uint8_t` and `uint32_t`. But specially, when the data width is 24-bit, the data buffer should be aligned with 3-byte (i.e., every 3 bytes stands for a 24-bit data in one slot). Additionally, `i2s_chan_config_t::dma_frame_num`, `i2s_std_clk_config_t::mclk_multiple`, and the writing buffer size should be the multiple of 3, otherwise the data on the line or the sample rate will be incorrect.

```c
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t tx_handle;

/* Get the default channel configuration by the helper macro. 
 * This helper macro is defined in `i2s_common.h` and shared by all the I2S 
 * communication modes. 
 * It can help to specify the I2S role and port ID */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_MASTER);

/* Allocate a new TX channel and get the handle of this channel */
i2s_new_channel(&chan_cfg, &tx_handle, NULL);

/* Setting the configurations, the slot configuration and clock configuration can 
 * be generated by the macros 
 * These two helper macros are defined in `i2s_std.h` which can only be used in 
 * STD mode. 
 * They can help to specify the slot and clock configurations for initialization 
 * or updating */
i2s_std_config_t std_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(48000),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT, I2S_SLOT_MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_4,
        .ws = GPIO_NUM_5,
        .dout = GPIO_NUM_18,
        .din = I2S_GPIO_UNUSED,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },

(continues on next page)
```
Chapter 2. API Reference

(continued from previous page)

};

/* Initialize the channel */
i2s_channel_init_std_mode(tx_handle, &std_cfg);

/* Before writing data, start the TX channel first */
i2s_channel_enable(tx_handle);
i2s_channel_write(tx_handle, src_buf, bytes_to_write, bytes_written, ticks_to_wait);

/* If the configurations of slot or clock need to be updated, */
* stop the channel first and then update it */
// i2s_channel_disable(tx_handle);
// std_cfg.slot_cfg.slot_mode = I2S_SLOT_MODE_MONO; // Default is stereo
// i2s_channel_reconfig_std_slot(tx_handle, &std_cfg.slot_cfg);
// std_cfg.clk_cfg.sample_rate_hz = 96000;
// i2s_channel_reconfig_std_clock(tx_handle, &std_cfg.clk_cfg);

/* Have to stop the channel before deleting it */
i2s_channel_disable(tx_handle);
/* If the handle is not needed any more, delete it to release the channel_*/
resources */
i2s_del_channel(tx_handle);

STD RX Mode  Taking 16-bit data width for example, when the data on the line are:

| WS low | WS high | WS low | WS high | WS low | WS high | WS low | WS high | ...
|-------|---------|--------|---------|--------|---------|--------|---------|-----
| 0x0001 | 0x0002  | 0x0003 | 0x0004  | 0x0005 | 0x0006  | 0x0007 | 0x0008  | ...

Here is the table of the data received in the buffer with different i2s_std_slot_config_t::slot_mode and i2s_std_slot_config_t::slot_mask.

<table>
<thead>
<tr>
<th>data bit width</th>
<th>slot mode</th>
<th>slot mask</th>
<th>data</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>mono</td>
<td>left</td>
<td>0x0001</td>
<td>0x0003</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td></td>
<td>0x0005</td>
<td>0x0007</td>
</tr>
<tr>
<td></td>
<td>stereo</td>
<td>any</td>
<td>0x0002</td>
<td>0x0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0006</td>
<td>0x0008</td>
</tr>
</tbody>
</table>

Note: 8-bit, 24-bit, and 32-bit are similar as 16-bit, where the data bit-width in the receiving buffer is equal to the data bit-width on the line. Additionally, when using 24-bit data width, i2s_chan_config_t::dma_frame_num, i2s_std_clk_config_t::mclk_multiple, and the receiving buffer size should be the multiple of 3, otherwise the data on the line or the sample rate will be incorrect.

#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t rx_handle;
/* Get the default channel configuration by helper macro. */
* This helper macro is defined in `i2s_common.h` and shared by all the I2S_*
communication modes.
* It can help to specify the I2S role and port ID */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_*/
*MASTER);
/* Allocate a new RX channel and get the handle of this channel */
i2s_new_channel(&chan_cfg, NULL, &rx_handle);

(continues on next page)
/* Setting the configurations, the slot configuration and clock configuration can be generated by the macros. 
These two helper macros are defined in `i2s_std.h` which can only be used in STD mode. 
They can help to specify the slot and clock configurations for initialization or updating. */

```c
i2s_std_config_t std_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(48000),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT, I2S_SLOT_MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_4,
        .ws = GPIO_NUM_5,
        .dout = I2S_GPIO_UNUSED,
        .din = GPIO_NUM_19,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};
```

/* Initialize the channel */
```c
i2s_channel_init_std_mode(rx_handle, &std_cfg);
```

/* Before reading data, start the RX channel first */
```c
i2s_channel_enable(rx_handle);
```

/* Have to stop the channel before deleting it */
```c
i2s_channel_disable(rx_handle);
```

/* If the handle is not needed any more, delete it to release the channel resources */
```c
i2s_del_channel(rx_handle);
```

**Full-duplex**

Full-duplex mode registers TX and RX channel in an I2S port at the same time, and the channels share the BCLK and WS signals. Currently, STD and TDM communication modes supports full-duplex mode in the following way, but PDM full-duplex is not supported because due to different PDM TX and RX clocks.

Note that one handle can only stand for one channel. Therefore, it is still necessary to configure the slot and clock for both TX and RX channels one by one.

Here is an example of how to allocate a pair of full-duplex channels:

```c
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t tx_handle;
i2s_chan_handle_t rx_handle;

/* Allocate a pair of I2S channel */
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_MASTER);
/* Allocate for TX and RX channel at the same time, then they will work in full-duplex mode */
i2s_new_channel(&chan_cfg, &tx_handle, &rx_handle);
/* Set the configurations for BOTH TWO channels, since TX and RX channel have to be same in full-duplex mode */
```
Chapter 2. API Reference

(continued from previous page)

```
i2s_std_config_t std_cfg = {  
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(32000),  
    .slot_cfg = I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO), 
    .gpio_cfg = {  
        .mclk = I2S_GPIO_UNUSED,  
        .bclk = GPIO_NUM_4,  
        .ws = GPIO_NUM_5,  
        .dout = GPIO_NUM_18,  
        .din = GPIO_NUM_19,  
        .invert_flags = {  
            .mclk_inv = false,  
            .bclk_inv = false,  
            .ws_inv = false,  
        },  
    },
};
i2s_channel_init_std_mode(tx_handle, &std_cfg);
i2s_channel_init_std_mode(rx_handle, &std_cfg);
i2s_channel_enable(tx_handle);
i2s_channel_enable(rx_handle);
```

**Simplex Mode** To allocate a channel handle in simplex mode, `i2s_new_channel()` should be called for each channel. The clock and GPIO pins of TX/RX channel on ESP32-S2 are not independent, so the TX and RX channel cannot coexist on the same I2S port in simplex mode.

```
#include "driver/i2s_std.h"
#include "driver/gpio.h"

i2s_chan_handle_t tx_handle;
i2s_chan_handle_t rx_handle;

i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO, I2S_ROLE_MASTER);
i2s_new_channel(chan_cfg, &tx_handle, NULL);
i2s_std_config_t std_tx_cfg = {  
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(48000),  
    .slot_cfg = I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO), 
    .gpio_cfg = {  
        .mclk = GPIO_NUM_0,  
        .bclk = GPIO_NUM_4,  
        .ws = GPIO_NUM_5,  
        .dout = GPIO_NUM_18,  
        .din = I2S_GPIO_UNUSED,  
        .invert_flags = {  
            .mclk_inv = false,  
            .bclk_inv = false,  
            .ws_inv = false,  
        },  
    },
};
/* Initialize the channel */
i2s_channel_init_std_mode(tx_handle, &std_tx_cfg);
i2s_channel_enable(tx_handle);
/* RX channel will be registered on another I2S, if no other available I2S unit found */
```

(continues on next page)
* it will return ESP_ERR_NOT_FOUND */
i2s_new_channel(schan_cfg, NULL, &rx_handle);
i2s_std_config_t std_rx_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(16000),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT, I2S_SLOT_MODE_STEREO),
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_6,
        .ws = GPIO_NUM_7,
        .dout = I2S_GPIO_UNUSED,
        .din = GPIO_NUM_19,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};
i2s_channel_init_std_mode(rx_handle, &std_rx_cfg);
i2s_channel_enable(rx_handle);

Application Notes

How to Prevent Data Lost  For applications that need a high frequency sample rate, the massive data throughput may cause data lost. Users can receive data lost event by registering the ISR callback function to receive the event queue:

```c
static IRAM_ATTR bool i2s_rx_queue_overflow_callback(i2s_chan_handle_t handle, i2s_event_data_t *event, void *user_ctx)
{
    // handle RX queue overflow event ...
    return false;
}
```

Please follow these steps to prevent data lost:

1. Determine the interrupt interval. Generally, when data lost happens, the bigger the interval, the better, which helps to reduce the interrupt times. This means `dma_frame_num` should be as big as possible while the DMA buffer size is below the maximum value of 4092. The relationships are:

```c
interrupt_interval(unit: sec) = dma_frame_num / sample_rate
dma_buffer_size = dma_frame_num * slot_num * data_bit_width / 8 <= 4092
```

2. Determine `dma_desc_num`. `dma_desc_num` is decided by the maximum time of `i2s_channel_read` polling cycle. All the received data is supposed to be stored between two `i2s_channel_read`. This cycle can be measured by a timer or an outputting GPIO signal. The relationship is:

```c
dma_desc_num > polling_cycle / interrupt_interval
```

3. Determine the receiving buffer size. The receiving buffer offered by users in `i2s_channel_read` should be able to take all the data in all DMA buffers, which means that it should be larger than the total size of all the DMA buffers:
recv_buffer_size > dma_desc_num * dma_buffer_size

For example, if there is an I2S application, and the known values are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_rate</td>
<td>144000 Hz</td>
</tr>
<tr>
<td>data_bit_width</td>
<td>32 bits</td>
</tr>
<tr>
<td>slot_num</td>
<td>2</td>
</tr>
<tr>
<td>polling_cycle</td>
<td>10 ms</td>
</tr>
</tbody>
</table>

Then the parameters dma_frame_num, dma_desc_num, and recv_buf_size can be calculated as follows:

\[
\text{dma_frame_num} \times \text{slot_num} \times \text{data_bit_width} / 8 \leq \text{dma_buffer_size} \leq 4092 \\
\text{interrupt_interval} = \text{dma_frame_num} / \text{sample_rate} = 511 / 144000 = 0.003549 \text{ s} = 3.549 \text{ ms} \\
\text{dma_desc_num} > \frac{\text{polling_cycle}}{\text{interrupt_interval}} = \text{cell}(10 / 3.549) = \text{cell}(2.818)_< \\
\text{recv_buffer_size} > \text{dma_desc_num} \times \text{dma_buffer_size} = 3 \times 4092 = 12276 \text{ bytes}
\]

**API Reference**

**Standard Mode**

**Header File**

- components/driver/i2s/include/driver/i2s_std.h
- This header file can be included with:
  ```
  #include "driver/i2s_std.h"
  ```
- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
  ```
  REQUIRES driver
  ```
  or
  ```
  PRIV_REQUIRES driver
  ```

**Functions**

`esp_err_t i2s_channel_init_std_mode(i2s_chan_handle_t handle, const i2s_std_config_t *std_cfg)`

Initialize I2S channel to standard mode.

**Note:** Only allowed to be called when the channel state is REGISTERED, (i.e., channel has been allocated, but not initialized) and the state will be updated to READY if initialization success, otherwise the state will return to REGISTERED.

**Parameters**

- `handle` - [in] I2S channel handler
- `std_cfg` - [in] Configurations for standard mode, including clock, slot and GPIO. The clock configuration can be generated by the helper macro `I2S_STD_CLK_DEFAULT_CONFIG`. The slot configuration can be generated by the helper macro `I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG`, `I2S_STD_PCM_SLOT_DEFAULT_CONFIG` or `I2S_STD_MSB_SLOT_DEFAULT_CONFIG`.

**Returns**

- ESP_OK Initialize successfully
• ESP_ERR_NO_MEM No memory for storing the channel information
• ESP_ERR_INVALID_ARG NULL pointer or invalid configuration
• ESP_ERR_INVALID_STATE This channel is not registered

```c
esp_err_t i2s_channel_reconfig_std_clock (i2s_chan_handle_t handle, const i2s_std_clk_config_t *clk_cfg)
```

Reconfigure the I2S clock for standard mode.

**Note:** Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won’t change the state. `i2s_channel_disable` should be called before calling this function if I2S has started.

**Parameters**
- `handle` - [in] I2S channel handler
- `clk_cfg` - [in] Standard mode clock configuration, can be generated by `I2S_STD_CLK_DEFAULT_CONFIG`

**Returns**
- ESP_OK Set clock successfully
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not standard mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

```c
esp_err_t i2s_channel_reconfig_std_slot (i2s_chan_handle_t handle, const i2s_std_slot_config_t *slot_cfg)
```

Reconfigure the I2S slot for standard mode.

**Note:** Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not started this function won’t change the state. `i2s_channel_disable` should be called before calling this function if I2S has started.

**Parameters**
- `handle` - [in] I2S channel handler
- `slot_cfg` - [in] Standard mode slot configuration, can be generated by `I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG`, `I2S_STD_PCM_SLOT_DEFAULT_CONFIG` and `I2S_STD_MSB_SLOT_DEFAULT_CONFIG`.

**Returns**
- ESP_OK Set clock successfully
- ESP_ERR_NO_MEM No memory for DMA buffer
- ESP_ERR_INVALID_ARG NULL pointer, invalid configuration or not standard mode
- ESP_ERR_INVALID_STATE This channel is not initialized or not stopped

```c
esp_err_t i2s_channel_reconfig_std_gpio (i2s_chan_handle_t handle, const i2s_std_gpio_config_t *gpio_cfg)
```

Reconfigure the I2S GPIO for standard mode.

**Note:** Only allowed to be called when the channel state is READY, i.e., channel has been initialized, but not
started this function won’t change the state. \texttt{i2s\_channel\_disable} should be called before calling this function if I2S has started.

\begin{center}
\textbf{Note:} The input channel handle has to be initialized to standard mode, i.e., \texttt{i2s\_channel\_init\_std\_mode} has been called before reconfiguring
\end{center}

\begin{description}
\item[Parameters]
  \begin{itemize}
  \item \texttt{handle} – [in] I2S channel handler
  \item \texttt{gpio\_cfg} – [in] Standard mode GPIO configuration, specified by user
  \end{itemize}

\item[Returns]
  \begin{itemize}
  \item ESP\_OK Set clock successfully
  \item ESP\_ERR\_INVALID\_ARG NULL pointer, invalid configuration or not standard mode
  \item ESP\_ERR\_INVALID\_STATE This channel is not initialized or not stopped
  \end{itemize}
\end{description}

\textbf{Structures}

\begin{verbatim}
struct i2s\_std\_slot\_config\_t
I2S slot configuration for standard mode.

Public Members

\end{verbatim}

\begin{verbatim}
i2s\_data\_bit\_width\_t \texttt{data\_bit\_width}
I2S sample data bit width (valid data bits per sample)

i2s\_slot\_bit\_width\_t \texttt{slot\_bit\_width}
I2S slot bit width (total bits per slot)

i2s\_slot\_mode\_t \texttt{slot\_mode}
Set mono or stereo mode with I2S\_SLOT\_MODE\_MONO or I2S\_SLOT\_MODE\_STEREO In TX direction, mono means the written buffer contains only one slot data and stereo means the written buffer contains both left and right data

i2s\_std\_slot\_mask\_t \texttt{slot\_mask}
Select the left, right or both slot

uint32\_t \texttt{ws\_width}
WS signal width (i.e. the number of BCLK ticks that WS signal is high)

bool \texttt{ws\_pol}
WS signal polarity, set true to enable high lever first

bool \texttt{bit\_shift}
Set to enable bit shift in Philips mode

bool \texttt{msb\_right}
Set to place right channel data at the MSB in the FIFO

struct i2s\_std\_clk\_config\_t
I2S clock configuration for standard mode.
\end{verbatim}
Public Members

```c
uint32_t sample_rate_hz

I2S sample rate
```

```c
i2s_clock_src_t clk_src
```
Choose clock source, see `soc_periph_i2s_clk_src_t` for the supported clock sources. selected `I2S_CLK_SRC_EXTERNAL` (if supports) to enable the external source clock input via MCLK pin.

```c
i2s_mclk_multiple_t mclk_multiple
```
The multiple of MCLK to the sample rate Default is 256 in the helper macro, it can satisfy most of cases, but please set this field a multiple of 3 (like 384) when using 24-bit data width, otherwise the sample rate might be inaccurate

```c
struct i2s_std_gpio_config_t
```
I2S standard mode GPIO pins configuration.

Public Members

```c
gpio_num_t mclk
```
MCK pin, output by default, input if the clock source is selected to `I2S_CLK_SRC_EXTERNAL`

```c
gpio_num_t bclk
```
BCK pin, input in slave role, output in master role

```c
gpio_num_t ws
```
WS pin, input in slave role, output in master role

```c
gpio_num_t dout
```
DATA pin, output

```c
gpio_num_t din
```
DATA pin, input

```c
uint32_t mclk_inv
```
Set 1 to invert the MCLK input/output

```c
uint32_t bclk_inv
```
Set 1 to invert the BCLK input/output

```c
uint32_t ws_inv
```
Set 1 to invert the WS input/output

```c
struct i2s_std_gpio_config_t::[anonymous] invert_flags
```
GPIO pin invert flags

```c
struct i2s_std_config_t
```
I2S standard mode major configuration that including clock/slot_GPIO configuration.
Public Members

\textit{i2s\_std\_clk\_config\_t} \texttt{clk\_cfg}

Standard mode clock configuration, can be generated by macro \texttt{I2S\_STD\_CLK\_DEFAULT\_CONFIG}

\textit{i2s\_std\_slot\_config\_t} \texttt{slot\_cfg}

Standard mode slot configuration, can be generated by macros \texttt{I2S\_STD\_[mode]\_SLOT\_DEFAULT\_CONFIG}, [mode] can be replaced with PHILIPS/MSB/PCM

\textit{i2s\_std\_gpio\_config\_t} \texttt{gpio\_cfg}

Standard mode GPIO configuration, specified by user

Macros

\texttt{I2S\_STD\_PHILIPS\_SLOT\_DEFAULT\_CONFIG}(bits\_per\_sample, mono\_or\_stereo)

Philips format in 2 slots.

This file is specified for I2S standard communication mode Features:

• Philips/MSB/PCM are supported in standard mode
• Fixed to 2 slots

Parameters

• \texttt{bits\_per\_sample} - I2S data bit width
• \texttt{mono\_or\_stereo} - I2S\_SLOT\_MODE\_MONO or I2S\_SLOT\_MODE\_STEREO

\texttt{I2S\_STD\_PCM\_SLOT\_DEFAULT\_CONFIG}(bits\_per\_sample, mono\_or\_stereo)

PCM (short) format in 2 slots.

\textbf{Note:} PCM (long) is same as Philips in 2 slots

Parameters

• \texttt{bits\_per\_sample} - I2S data bit width
• \texttt{mono\_or\_stereo} - I2S\_SLOT\_MODE\_MONO or I2S\_SLOT\_MODE\_STEREO

\texttt{I2S\_STD\_MSB\_SLOT\_DEFAULT\_CONFIG}(bits\_per\_sample, mono\_or\_stereo)

MSB format in 2 slots.

Parameters

• \texttt{bits\_per\_sample} - I2S data bit width
• \texttt{mono\_or\_stereo} - I2S\_SLOT\_MODE\_MONO or I2S\_SLOT\_MODE\_STEREO

\texttt{I2S\_STD\_CLK\_DEFAULT\_CONFIG}(rate)

I2S default standard clock configuration.

\textbf{Note:} Please set the mclk\_multiple to I2S\_MCLK\_MULTIPLE\_384 while using 24 bits data width Otherwise the sample rate might be imprecise since the BCLK division is not a integer

Parameters

• \texttt{rate} - sample rate

\textbf{I2S Driver}
Header File

- components/driver/i2s/include/driver/i2s_common.h
- This header file can be included with:

```c
#include "driver/i2s_common.h"
```

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```text
REQUIRES driver
```

or

```text
PRIV_REQUIRES driver
```

Functions

**`esp_err_t i2s_new_channel`**

```c
(const i2s_chan_config_t *chan_cfg,
 i2s_chan_handle_t *ret_tx_handle,
 i2s_chan_handle_t *ret_rx_handle)
```

Allocate new I2S channel(s)

**Note:** The new created I2S channel handle will be REGISTERED state after it is allocated successfully.

**Note:** When the port id in channel configuration is I2S_NUM_AUTO, driver will allocate I2S port automatically on one of the I2S controller, otherwise driver will try to allocate the new channel on the selected port.

**Note:** If both tx_handle and rx_handle are not NULL, it means this I2S controller will work at full-duplex mode, the RX and TX channels will be allocated on a same I2S port in this case. Note that some configurations of TX/RX channel are shared on ESP32 and ESP32S2, so please make sure they are working at same condition and under same status(start/stop). Currently, full-duplex mode can’t guarantee TX/RX channels write/read synchronously, they can only share the clock signals for now.

**Note:** If tx_handle OR rx_handle is NULL, it means this I2S controller will work at simplex mode. For ESP32 and ESP32S2, the whole I2S controller (i.e. both RX and TX channel) will be occupied, even if only one of RX or TX channel is registered. For the other targets, another channel on this controller will still available.

**Parameters**

- `chan_cfg` - in] I2S controller channel configurations
- `ret_tx_handle` - [out] I2S channel handler used for managing the sending channel(optional)
- `ret_rx_handle` - [out] I2S channel handler used for managing the receiving channel(optional)

**Returns**

- `ESP_OK` Allocate new channel(s) success
- `ESP_ERR_NOT_SUPPORTED` The communication mode is not supported on the current chip
- `ESP_ERR_INVALID_ARG` NULL pointer or illegal parameter in `i2s_chan_config_t`
- `ESP_ERR_NOT_FOUND` No available I2S channel found

**`esp_err_t i2s_del_channel`**

```c
(i2s_chan_handle_t handle)
```

Delete the I2S channel.
**Chapter 2. API Reference**

**Note:** Only allowed to be called when the I2S channel is at REGISTERED or READY state (i.e., it should stop before deleting it).

**Note:** Resource will be free automatically if all channels in one port are deleted.

### Parameters
- **handle** - [in] I2S channel handler
  - ESP_OK Delete successfully
  - ESP_ERR_INVALID_ARG NULL pointer

**esp_err_t i2s_channel_get_info** *(i2s_chan_handle_t handle, i2s_chan_info_t *chan_info)*
Get I2S channel information.

**Parameters**
- **handle** - [in] I2S channel handler
- **chan_info** - [out] I2S channel basic information

**Returns**
- ESP_OK Get I2S channel information success
- ESP_ERR_NOT_FOUND The input handle doesn’t match any registered I2S channels, it may not an I2S channel handle or not available any more
- ESP_ERR_INVALID_ARG The input handle or chan_info pointer is NULL

**esp_err_t i2s_channel_enable** *(i2s_chan_handle_t handle)*
Enable the I2S channel.

**Note:** Only allowed to be called when the channel state is READY, (i.e., channel has been initialized, but not started) the channel will enter RUNNING state once it is enabled successfully.

**Note:** Enable the channel can start the I2S communication on hardware. It will start outputting BCLK and WS signal. For MCLK signal, it will start to output when initialization is finished.

### Parameters
- **handle** - [in] I2S channel handler
  - ESP_OK Start successfully
  - ESP_ERR_INVALID_ARG NULL pointer
  - ESP_ERR_INVALID_STATE This channel has not initialized or already started

**esp_err_t i2s_channel_disable** *(i2s_chan_handle_t handle)*
Disable the I2S channel.

**Note:** Only allowed to be called when the channel state is RUNNING, (i.e., channel has been started) the channel will enter READY state once it is disabled successfully.

**Note:** Disable the channel can stop the I2S communication on hardware. It will stop BCLK and WS signal but not MCLK signal.

### Parameters
- **handle** - [in] I2S channel handler

**Returns**
- ESP_OK Stop successfully
- ESP_ERR_INVALID_ARG NULL pointer
- ESP_ERR_INVALID_STATE This channel has not stated
**esp_err_t i2s_channel_preload_data(i2s_chan_handle_t tx_handle, const void *src, size_t size, size_t *bytes_loaded)**

Preload the data into TX DMA buffer.

**Note:** Only allowed to be called when the channel state is READY, (i.e., channel has been initialized, but not started)

**Note:** As the initial DMA buffer has no data inside, it will transmit the empty buffer after enabled the channel, this function is used to preload the data into the DMA buffer, so that the valid data can be transmitted immediately after the channel is enabled.

**Note:** This function can be called multiple times before enabling the channel, the buffer that loaded later will be concatenated behind the former loaded buffer. But when all the DMA buffers have been loaded, no more data can be preload then, please check the `bytes_loaded` parameter to see how many bytes are loaded successfully, when the `bytes_loaded` is smaller than the `size`, it means the DMA buffers are full.

**Parameters**
- `tx_handle` – [in] I2S TX channel handler
- `src` – [in] The pointer of the source buffer to be loaded
- `size` – [in] The source buffer size
- `bytes_loaded` – [out] The bytes that successfully been loaded into the TX DMA buffer

**Returns**
- ESP_OK Load data successful
- ESP_ERR_INVALID_ARG NULL pointer or not TX direction
- ESP_ERR_INVALID_STATE This channel has not stated

**esp_err_t i2s_channel_write(i2s_chan_handle_t handle, const void *src, size_t size, size_t *bytes_written, uint32_t timeout_ms)**

I2S write data.

**Note:** Only allowed to be called when the channel state is RUNNING, (i.e., TX channel has been started and is not writing now) but the RUNNING only stands for the software state, it doesn’t mean there is no the signal transporting on line.

**Parameters**
- `handle` – [in] I2S channel handler
- `src` – [in] The pointer of sent data buffer
- `size` – [in] Max data buffer length
- `bytes_written` – [out] Byte number that actually be sent, can be NULL if not needed
- `timeout_ms` – [in] Max block time

**Returns**
- ESP_OK Write successfully
- ESP_ERR_INVALID_ARG NULL pointer or this handle is not TX handle
- ESP_ERR_TIMEOUT Writing timeout, no writing event received from ISR within ticks_to_wait
- ESP_ERR_INVALID_STATE I2S is not ready to write

**esp_err_t i2s_channel_read(i2s_chan_handle_t handle, void *dest, size_t size, size_t *bytes_read, uint32_t timeout_ms)**

I2S read data.
**Note:** Only allowed to be called when the channel state is RUNNING but the RUNNING only stands for the software state, it doesn’t mean there is no the signal transporting on line.

### Parameters
- `handle` - [in] I2S channel handler
- `dest` - [in] The pointer of receiving data buffer
- `size` - [in] Max data buffer length
- `bytes_read` - [out] Byte number that actually be read, can be NULL if not needed
- `timeout_ms` - [in] Max block time

### Returns
- ESP_OK Read successfully
- ESP_ERR_INVALID_ARG NULL pointer or this handle is not RX handle
- ESP_ERR_TIMEOUT Reading timeout, no reading event received from ISR within ticks_to_wait
- ESP_ERR_INVALID_STATE I2S is not ready to read

```c
esp_err_t i2s_channel_register_event_callback(i2s_chan_handle_t handle, const i2s_event_callbacks_t *callbacks, void *user_data)
```

Set event callbacks for I2S channel.

**Note:** Only allowed to be called when the channel state is REGISTERED / READY, (i.e., before channel starts)

**Note:** User can deregister a previously registered callback by calling this function and setting the callback member in the callbacks structure to NULL.

**Note:** When CONFIG_I2S_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM or internal RAM as well.

### Parameters
- `handle` - [in] I2S channel handler
- `callbacks` - [in] Group of callback functions
- `user_data` - [in] User data, which will be passed to callback functions directly

### Returns
- ESP_OK Set event callbacks successfully
- ESP_ERR_INVALID_ARG Set event callbacks failed because of invalid argument
- ESP_ERR_INVALID_STATE Set event callbacks failed because the current channel state is not REGISTERED or READY

### Structures

```c
struct i2s_event_callbacks_t
```

Group of I2S callbacks.

**Note:** The callbacks are all running under ISR environment
Note: When CONFIG_I2S_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

Public Members

i2s_isr_callback_t on_recv
Callback of data received event, only for RX channel The event data includes DMA buffer address and size that just finished receiving data

i2s_isr_callback_t on_recv_q_ovf
Callback of receiving queue overflowed event, only for RX channel The event data includes buffer size that has been overwritten

i2s_isr_callback_t on_sent
Callback of data sent event, only for TX channel The event data includes DMA buffer address and size that just finished sending data

i2s_isr_callback_t on_send_q_ovf
Callback of sending queue overflowed event, only for TX channel The event data includes buffer size that has been overwritten

struct i2s_chan_config_t
I2S controller channel configuration.

Public Members

i2s_port_t id
I2S port id

i2s_role_t role
I2S role, I2S_ROLE_MASTER or I2S_ROLE_SLAVE

uint32_t dma_desc_num
I2S DMA buffer number, it is also the number of DMA descriptor

uint32_t dma_frame_num
I2S frame number in one DMA buffer. One frame means one-time sample data in all slots, it should be the multiple of 3 when the data bit width is 24.

bool auto_clear
Set to auto clear DMA TX buffer, I2S will always send zero automatically if no data to send

int intr_priority
I2S interrupt priority, range [0, 7], if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

struct i2s_chan_info_t
I2S channel information.
Public Members

\textit{i2s\_port\_t id}
I2S port id

\textit{i2s\_role\_t role}
I2S role, I2S\_ROLE\_MASTER or I2S\_ROLE\_SLAVE

\textit{i2s\_dir\_t dir}
I2S channel direction

\textit{i2s\_comm\_mode\_t mode}
I2S channel communication mode

\textit{i2s\_chan\_handle\_t pair\_chan}
I2S pair channel handle in duplex mode, always NULL in simplex mode

Macros

\textit{I2S\_CHANNEL\_DEFAULT\_CONFIG} (i2s\_num, i2s\_role)
get default I2S property

\textit{I2S\_GPIO\_UNUSED}
Used in i2s\_gpio\_config\_t for signals which are not used

I2S Types

Header File

- components/driver/i2s/include/driver/i2s\_types.h
- This header file can be included with:

\begin{verbatim}
#include "driver/i2s\_types.h"
\end{verbatim}

- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

\begin{verbatim}
REQUIRES driver
\end{verbatim}

or

\begin{verbatim}
PRIV_REQUIRES driver
\end{verbatim}

Structures

\texttt{struct i2s\_event\_data\_t}
Event structure used in I2S event queue.

Public Members

void *data
The pointer of DMA buffer that just finished sending or receiving for \texttt{on\_recv} and \texttt{on\_sent} callback
NULL for \texttt{on\_recv\_q\_ovf} and \texttt{on\_send\_q\_ovf} callback
size_t size

The buffer size of DMA buffer when success to send or receive, also the buffer size that dropped when queue overflow. It is related to the dma_frame_num and data_bit_width, typically it is fixed when data_bit_width is not changed.

Type Definitions

typedef struct i2s_channel_obj_t *i2s_chan_handle_t

I2S channel object handle, the control unit of the I2S driver

typedef bool (*i2s_isr_callback_t)(i2s_chan_handle_t handle, i2s_event_data_t *event, void *user_ctx)

I2S event callback.

Param handle [in] I2S channel handle, created from i2s_new_channel()
Param event [in] I2S event data
Param user_ctx [in] User registered context, passed from i2s_channel_register_event_callback()

Return Whether a high priority task has been waken up by this callback function

Enumerations

enum i2s_port_t

I2S controller port number, the max port number is (SOC_I2S_NUM -1).

Values:

enumerator I2S_NUM_0

I2S controller port 0

enumerator I2S_NUM_AUTO

Select whichever port is available

enum i2s_comm_mode_t

I2S controller communication mode.

Values:

enumerator I2S_COMM_MODE_STD

I2S controller using standard communication mode, support Philips/MSB/PCM format

enumerator I2S_COMM_MODE_NONE

Unspecified I2S controller mode

enum i2s_mclk_multiple_t

The multiple of MCLK to sample rate.

Values:

enumerator I2S_MCLK_MULTIPLE_128

MCLK = sample_rate * 128

enumerator I2S_MCLK_MULTIPLE_256

MCLK = sample_rate * 256
enumerator **I2S_MCLK_MULTIPLE_384**  
MCLK = sample_rate * 384

enumerator **I2S_MCLK_MULTIPLE_512**  
MCLK = sample_rate * 512

**Header File**

- components/hal/include/hal/i2s_types.h
- This header file can be included with:

```c
#include "hal/i2s_types.h"
```

**Type Definitions**

typedef `soc_periph_i2s_clk_src_t` **i2s_clock_src_t**  
I2S clock source

**Enumerations**

enum **i2s_slot_mode_t**  
I2S channel slot mode.

Values:

- enumerator **I2S_SLOT_MODE_MONO**  
  I2S channel slot format mono, transmit same data in all slots for tx mode, only receive the data in the first slots for rx mode.

- enumerator **I2S_SLOT_MODE_STEREO**  
  I2S channel slot format stereo, transmit different data in different slots for tx mode, receive the data in all slots for rx mode.

enum **i2s_dir_t**  
I2S channel direction.

Values:

- enumerator **I2S_DIR_RX**  
  I2S channel direction RX

- enumerator **I2S_DIR_TX**  
  I2S channel direction TX

enum **i2s_role_t**  
I2S controller role.

Values:

- enumerator **I2S_ROLE_MASTER**  
  I2S controller master role, belk and ws signal will be set to output
enumerator **I2S_ROLE_SLAVE**
I2S controller slave role, bclk and ws signal will be set to input

enum **i2s_data_bit_width_t**
Available data bit width in one slot.

*Values:*

enumerator **I2S_DATA_BIT_WIDTH_8BIT**
I2S channel data bit-width: 8

enumerator **I2S_DATA_BIT_WIDTH_16BIT**
I2S channel data bit-width: 16

enumerator **I2S_DATA_BIT_WIDTH_24BIT**
I2S channel data bit-width: 24

enumerator **I2S_DATA_BIT_WIDTH_32BIT**
I2S channel data bit-width: 32

enum **i2s_slot_bit_width_t**
Total slot bit width in one slot.

*Values:*

enumerator **I2S_SLOT_BIT_WIDTH_AUTO**
I2S channel slot bit-width equals to data bit-width

enumerator **I2S_SLOT_BIT_WIDTH_8BIT**
I2S channel slot bit-width: 8

enumerator **I2S_SLOT_BIT_WIDTH_16BIT**
I2S channel slot bit-width: 16

enumerator **I2S_SLOT_BIT_WIDTH_24BIT**
I2S channel slot bit-width: 24

enumerator **I2S_SLOT_BIT_WIDTH_32BIT**
I2S channel slot bit-width: 32

enum **i2s_std_slot_mask_t**
I2S slot select in standard mode.

*Note:* It has different meanings in tx/rx/mono/stereo mode, and it may have different behaviors on different targets For the details, please refer to the I2S API reference

*Values:*

enumerator **I2S_STD_SLOT_LEFT**
I2S transmits or receives left slot
### 2.5.13 LCD

**Introduction**

ESP chips can generate various kinds of timings that needed by common LCDs on the market, like SPI LCD, I80 LCD (a.k.a Intel 8080 parallel LCD), RGB/RGB LCD, I2C LCD, etc. The esp_lcd component is officially to support those LCDs with a group of universal APIs across chips.

**Functional Overview**

In esp_lcd, an LCD panel is represented by esp_lcd_panel_handle_t, which plays the role of an abstract frame buffer, regardless of the frame memory is allocated inside ESP chip or in external LCD controller. Based on the location of the frame buffer and the hardware connection interface, the LCD panel drivers are mainly grouped into the following categories:

- **Controller based LCD driver** involves multiple steps to get a panel handle, like bus allocation, IO device registration and controller driver install. The frame buffer is located in the controller’s internal GRAM (Graphical RAM). ESP-IDF provides only a limited number of LCD controller drivers out of the box (e.g., ST7789, SSD1306), More Controller Based LCD Drivers are maintained in the Espressif Component Registry [here](https://components.espressif.com/).
- **SPI Interfaced LCD** describes the steps to install the SPI LCD IO driver and then get the panel handle.
- **I2C Interfaced LCD** describes the steps to install the I2C LCD IO driver and then get the panel handle.
- **I80 Interfaced LCD** describes the steps to install the I80 LCD IO driver and then get the panel handle.
- **LCD Panel Basic Operations** - provides a set of APIs to operate the LCD panel, like turning on/off the display, setting the orientation, etc. These operations are common for either controller-based LCD panel driver or RGB LCD panel driver.

**SPI Interfaced LCD**

1. Create an SPI bus. Please refer to [SPI Master API doc](https://docs.espressif.com) for more details.
Chapter 2. API Reference

```c
spi_bus_config_t buscfg = {
    .sclk_io_num = EXAMPLE_PIN_NUM_SCLK,
    .mosi_io_num = EXAMPLE_PIN_NUM_MOSI,
    .miso_io_num = EXAMPLE_PIN_NUM_MISO,
    .quadwp_io_num = -1, // Quad SPI LCD driver is not yet supported
    .quadhd_io_num = -1, // Quad SPI LCD driver is not yet supported
    .max_transfer_sz = EXAMPLE_LCD_H_RES * 80 * sizeof(uint16_t), // transfer 80 lines of pixels (assume pixel is RGB565) at most in one SPI transaction
};
ESP_ERROR_CHECK(spi_bus_initialize(LCD_HOST, &buscfg, SPI_DMA_CH_AUTO)); // Enable the DMA feature
```

2. Allocate an LCD IO device handle from the SPI bus. In this step, you need to provide the following information:

- `esp_lcd_panel_io_spi_config_t::dc_gpio_num`: Sets the GPIO number for the DC signal line (some LCD calls this RS line). The LCD driver uses this GPIO to switch between sending command and sending data.
- `esp_lcd_panel_io_spi_config_t::cs_gpio_num`: Sets the GPIO number for the CS signal line. The LCD driver uses this GPIO to select the LCD chip. If the SPI bus only has one device attached (i.e., this LCD), you can set the GPIO number to -1 to occupy the bus exclusively.
- `esp_lcd_panel_io_spi_config_t::pclk_hz`: sets the frequency of the pixel clock, in Hz. The value should not exceed the range recommended in the LCD spec.
- `esp_lcd_panel_io_spi_config_t::spi_mode`: sets the SPI mode. The LCD driver uses this mode to communicate with the LCD. For the meaning of the SPI mode, please refer to the SPI Master API doc.
- `esp_lcd_panel_io_spi_config_t::lcd_cmd_bits` and `esp_lcd_panel_io_spi_config_t::lcd_param_bits`: set the bit width of the command and parameter that recognized by the LCD controller chip. This is chip specific, you should refer to your LCD spec in advance.
- `esp_lcd_panel_io_spi_config_t::trans_queue_depth`: sets the depth of the SPI transaction queue. A bigger value means more transactions can be queued up, but it also consumes more memory.

```c
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_spi_config_t io_config = {
    .dc_gpio_num = EXAMPLE_PIN_NUM_LCD_DC,
    .cs_gpio_num = EXAMPLE_PIN_NUM_LCD_CS,
    .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,
    .lcd_param_bits = EXAMPLE_LCD_PARAM_BITS,
    .spi_mode = 0,
    .trans_queue_depth = 10,
};
// Attach the LCD to the SPI bus
ESP_ERROR_CHECK(esp_lcd_new_panel_io_spi((esp_lcd_spi_bus_handle_t)LCD_HOST, &io_config, &io_handle));
```

3. Install the LCD controller driver. The LCD controller driver is responsible for sending the commands and parameters to the LCD controller chip. In this step, you need to specify the SPI IO device handle that allocated in the last step, and some panel specific configurations:

- `esp_lcd_panel_dev_config_t::reset_gpio_num`: sets the LCD’s hardware reset GPIO number. If the LCD does not have a hardware reset pin, set this to -1.
- `esp_lcd_panel_dev_config_t::rgb_ele_order`: sets the R-G-B element order of each color data.
- `esp_lcd_panel_dev_config_t::bits_per_pixel`: sets the bit width of the pixel color data. The LCD driver uses this value to calculate the number of bytes to send to the LCD controller chip.
- `esp_lcd_panel_dev_config_t::data_endian`: specifies the data endian to be transmitted to the screen. No need to specify for color data within 1 byte, like RGB232. For drivers that do not support specifying data endian, this field would be ignored.
Chapter 2. API Reference

```c
esp_lcd_panel_handle_t panel_handle = NULL;
esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_PIN_NUM_RST,
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_BGR,
    .bits_per_pixel = 16,
};
// Create LCD panel handle for ST7789, with the SPI IO device handle
ESP_ERROR_CHECK(esp_lcd_new_panel_st7789(io_handle, &panel_config, &panel_handle));
```

I2C Interfaced LCD

1. Create I2C bus. Please refer to [I2C API doc](#) for more details.

```c
i2c_config_t i2c_conf = {
    .mode = I2C_MODE_MASTER, // I2C LCD is a master node
    .sda_io_num = EXAMPLE_PIN_NUM_SDA,
    .scl_io_num = EXAMPLE_PIN_NUM_SCL,
    .sda_pullup_en = GPIO_PULLUP_ENABLE,
    .scl_pullup_en = GPIO_PULLUP_ENABLE,
    .master.clk_speed = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
};
ESP_ERROR_CHECK(i2c_param_config(I2C_HOST, &i2c_conf));
ESP_ERROR_CHECK(i2c_driver_install(I2C_HOST, I2C_MODE_MASTER, 0, 0, 0, 0));
```

2. Allocate an LCD IO device handle from the I2C bus. In this step, you need to provide the following information:
   - `esp_lcd_panel_i2c_bus_config_t::dev_addr` sets the I2C device address of the LCD controller chip. The LCD driver uses this address to communicate with the LCD controller chip.
   - `esp_lcd_panel_i2c_config_t::lcd_cmd_bits` and `esp_lcd_panel_i2c_config_t::lcd_param_bits` set the bit width of the command and parameter that recognized by the LCD controller chip. This is chip specific, you should refer to your LCD spec in advance.

```c
esp_lcd_panel_io_i2c_config_t io_config = {
    .dev_addr = EXAMPLE_I2C_HW_ADDR,
    .control_phase_bytes = 1, // refer to LCD spec
    .dc_bit_offset = 6, // refer to LCD spec
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,
    .lcd_param_bits = EXAMPLE_LCD_CMD_BITS,
};
ESP_ERROR_CHECK(esp_lcd_new_panel_i2c((esp_lcd_i2c_bus_handle_t)I2C_HOST, &io_config, &io_handle));
```

3. Install the LCD controller driver. The LCD controller driver is responsible for sending the commands and parameters to the LCD controller chip. In this step, you need to specify the I2C IO device handle that allocated in the last step, and some panel specific configurations:
   - `esp_lcd_panel_dev_config_t::reset_gpio_num` sets the LCD’s hardware reset GPIO number. If the LCD does not have a hardware reset pin, set this to -1.
   - `esp_lcd_panel_dev_config_t::bits_per_pixel` sets the bit width of the pixel color data. The LCD driver uses this value to calculate the number of bytes to send to the LCD controller chip.

```c
esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_PIN_NUM_RST,
    .bits_per_pixel = 16,
};
```

(continues on next page)
I80 Interfaced LCD

1. Create I80 bus by `esp_lcd_new_i80_bus()`. You need to set up the following parameters for an Intel 8080 parallel bus:
   - `esp_lcd_i80_bus_config_t::clk_src` sets the clock source of the I80 bus. Note, the default clock source may be different between ESP targets.
   - `esp_lcd_i80_bus_config_t::wr_gpio_num` sets the GPIO number of the pixel clock (also referred as WR in some LCD spec).
   - `esp_lcd_i80_bus_config_t::dc_gpio_num` sets the GPIO number of the data/command select pin (also referred as RS in some LCD spec).
   - `esp_lcd_i80_bus_config_t::bus_width` sets the bit width of the data bus (only support 8 or 16).
   - `esp_lcd_i80_bus_config_t::data_gpio_nums` is the array of the GPIO number of the data bus. The number of GPIOs should be equal to the `esp_lcd_i80_bus_config_t::bus_width` value.
   - `esp_lcd_i80_bus_config_t::max_transfer_bytes` sets the maximum number of bytes that can be transferred in one transaction.

   ```c
   esp_lcd_i80_bus_handle_t i80_bus = NULL;
   esp_lcd_i80_bus_config_t bus_config = {
      .clk_src = LCD_CLK_SRC_DEFAULT,
      .dc_gpio_num = EXAMPLE_PIN_NUM_DC,
      .wr_gpio_num = EXAMPLE_PIN_NUM_PCLK,
      .data_gpio_nums = {EXAMPLE_PIN_NUMDATA0,
                         EXAMPLE_PIN_NUMDATA1,
                         EXAMPLE_PIN_NUMDATA2,
                         EXAMPLE_PIN_NUMDATA3,
                         EXAMPLE_PIN_NUMDATA4,
                         EXAMPLE_PIN_NUMDATA5,
                         EXAMPLE_PIN_NUMDATA6,
                         EXAMPLE_PIN_NUMDATA7,
      },
      .bus_width = 8,
      .max_transfer_bytes = EXAMPLE_LCD_H_RES * 100 * sizeof(uint16_t),
     /* transfer 100 lines of pixels (assume pixel is RGB565) at most in one transaction */
      .psram_trans_align = EXAMPLE_PSRAM_DATA_ALIGNMENT,
      .sram_trans_align = 4,
   };
   ESP_ERROR_CHECK(esp_lcd_new_i80_bus( &bus_config, &i80_bus));
   ```

2. Allocate an LCD IO device handle from the I80 bus. In this step, you need to provide the following information:
   - `esp_lcd_panel_io_i80_config_t::cs_gpio_num` sets the GPIO number of the chip select pin.
   - `esp_lcd_panel_io_i80_config_t::pclk_hz` sets the pixel clock frequency in Hz. Higher pixel clock frequency results in higher refresh rate, but may cause flickering if the DMA bandwidth is not sufficient or the LCD controller chip does not support high pixel clock frequency.
   - `esp_lcd_panel_io_i80_config_t::lcd_cmd_bits` and `esp_lcd_panel_io_i80_config_t::lcd_param_bits` set the bit width of the command and parameter that recognized by the LCD controller chip. This is chip specific, you should refer to your LCD spec in advance.
   - `esp_lcd_panel_io_i80_config_t::trans_queue_depth` sets the maximum number of transactions that can be queued in the LCD IO device. A bigger value means more
transactions can be queued up, but it also consumes more memory.

```c
esp_lcd_panel_io_handle_t io_handle = NULL;
esp_lcd_panel_io_i80_config_t io_config = {
    .cs_gpio_num = EXAMPLE_PIN_NUM_CS,
    .pclk_hz = EXAMPLE_LCD_PIXEL_CLOCK_HZ,
    .trans_queue_depth = 10,
    .dc_levels = {
        .dc_idle_level = 0,
        .dc_cmd_level = 0,
        .dc_dummy_level = 0,
        .dc_data_level = 1,
    },
    .lcd_cmd_bits = EXAMPLE_LCD_CMD_BITS,
    .lcd_param_bits = EXAMPLE_LCD_PARAM_BITS,
};
ESP_ERROR_CHECK(esp_lcd_new_panel_io_i80(i80_bus, &io_config, &io_handle));
```

3. Install the LCD controller driver. The LCD controller driver is responsible for sending the commands and parameters to the LCD controller chip. In this step, you need to specify the I80 IO device handle that allocated in the last step, and some panel specific configurations:

- `esp_lcd_panel_dev_config_t::bits_per_pixel` sets the bit width of the pixel color data. The LCD driver uses this value to calculate the number of bytes to send to the LCD controller chip.
- `esp_lcd_panel_dev_config_t::reset_gpio_num` sets the GPIO number of the reset pin. If the LCD controller chip does not have a reset pin, you can set this value to -1.
- `esp_lcd_panel_dev_config_t::rgb_ele_order` sets the color order of the pixel color data.

```c
esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = EXAMPLE_PIN_NUM_RST,
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB,
    .bits_per_pixel = 16,
};
ESP_ERROR.Check(esp_lcd_new_panel_st7789(io_handle, &panel_config, &panel_handle));
```

More Controller Based LCD Drivers

More LCD panel drivers and touch drivers are available in ESP-IDF Component Registry. The list of available and planned drivers with links is in this table.

**LCD Panel Basic Operations**

- `esp_lcd_panel_reset()` can reset the LCD panel.
- `esp_lcd_panel_init()` performs a basic initialization of the panel. To perform more manufacture specific initialization, please go to **Steps to Add Manufacture Specific Initialization**.
- Through combined use of `esp_lcd_panel_swap_xy()` and `esp_lcd_panel_mirror()`, you can rotate the LCD screen.
- `esp_lcd_panel_disp_on_off()` can turn on or off the LCD screen by cutting down the output path from the frame buffer to the LCD screen.
- `esp_lcd_panel_disp_sleep()` can reduce the power consumption of the LCD screen by entering the sleep mode. The internal frame buffer is still retained.
- `esp_lcd_panel_draw_bitmap()` is the most significant function, which does the magic to draw the user provided color buffer to the LCD screen, where the draw window is also configurable.
Steps to Add Manufacture Specific Initialization

The LCD controller drivers (e.g., st7789) in ESP-IDF only provide basic initialization in the `esp_lcd_panel_init()`, leaving the vast majority of settings to the default values. Some LCD modules need to set a bunch of manufacture specific configurations before it can display normally. These configurations usually include gamma, power voltage and so on. If you want to add manufacture specific initialization, please follow the steps below:

```c
esp_lcd_panel_reset(panel_handle);
esp_lcd_panel_init(panel_handle);
// set extra configurations e.g., gamma control
// with the underlying IO handle
// please consult your manufacture for special commands and corresponding values
esp_lcd_panel_io_tx_param(io_handle, GAMMA_CMD, {GAMMA_ARRAY}, N);
// turn on the display
esp_lcd_panel_disp_on_off(panel_handle, true);
```

Application Example

LCD examples are located under: `peripherals/lcd`:

- Universal SPI LCD example with SPI touch - `peripherals/lcd/spi_lcd_touch`
- Jpeg decoding and LCD display - `peripherals/lcd/tjpgd`
- i80 controller based LCD and LVGL animation UI - `peripherals/lcd/i80_controller`
- I2C interfaced OLED display scrolling text - `peripherals/lcd/i2c_oled`

API Reference

Header File

- `components/hal/include/hal/lcd_types.h`
- This header file can be included with:

  ```c
  #include "hal/lcd_types.h"
  ```

Type Definitions

typedef `soc_periph_lcd_clk_src_t` `lcd_clock_source_t`

  LCD clock source.

Enumerations

enum `lcd_rgb_data_endian_t`

  RGB data endian.

  `Values`:

  enumerator `LCD_RGB_DATA_ENDIAN_BIG`

    RGB data endian: MSB first

  enumerator `LCD_RGB_DATA_ENDIAN_LITTLE`

    RGB data endian: LSB first
enum **lcd_color_space_t**
   LCD color space.
   
   Values:
   
   enumerator **LCD_COLOR_SPACE_RGB**
   Color space: RGB
   
   enumerator **LCD_COLOR_SPACE_YUV**
   Color space: YUV

enum **lcd_color_range_t**
   LCD color range.
   
   Values:
   
   enumerator **LCD_COLOR_RANGE_LIMIT**
   Limited color range
   
   enumerator **LCD_COLOR_RANGE_FULL**
   Full color range

enum **lcd_yuv_sample_t**
   YUV sampling method.
   
   Values:
   
   enumerator **LCD_YUV_SAMPLE_422**
   YUV 4:2:2 sampling
   
   enumerator **LCD_YUV_SAMPLE_420**
   YUV 4:2:0 sampling
   
   enumerator **LCD_YUV_SAMPLE_411**
   YUV 4:1:1 sampling

enum **lcd_yuv_conv_std_t**
   The standard used for conversion between RGB and YUV.
   
   Values:
   
   enumerator **LCD_YUV_CONV_STD_BT601**
   YUV<>RGB conversion standard: BT.601
   
   enumerator **LCD_YUV_CONV_STD_BT709**
   YUV<>RGB conversion standard: BT.709

**Header File**
- components/esp_lcd/include/esp_lcd_types.h
- This header file can be included with:
# Chapter 2. API Reference

```
#include "esp_lcd_types.h"
```

- This header file is a part of the API provided by the `esp_lcd` component. To declare that your component depends on `esp_lcd`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_lcd
```

or

```
PRIV_REQUIRES esp_lcd
```

### Type Definitions

typedef struct esp_lcd_panel_io_t *esp_lcd_panel_io_handle_t

Type of LCD panel IO handle

typedef struct esp_lcd_panel_t *esp_lcd_panel_handle_t

Type of LCD panel handle

### Enumerations

eenum lcd_rgb_element_order_t

RGB element order.

*Values:*

- enumerator `LCD_RGB_ELEMENT_ORDER_RGB`
  RGB element order: RGB

- enumerator `LCD_RGB_ELEMENT_ORDER_BGR`
  RGB element order: BGR

### Header File

- components/esp_lcd/include/esp_lcd_panel_io.h

- This header file can be included with:

```
#include "esp_lcd_panel_io.h"
```

- This header file is a part of the API provided by the `esp_lcd` component. To declare that your component depends on `esp_lcd`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_lcd
```

or

```
PRIV_REQUIRES esp_lcd
```

### Functions

```c
esp_err_t esp_lcd_panel_io_rx_param(esp_lcd_panel_io_handle_t io, int lcd_cmd, void *param, size_t param_size)
```

Transmit LCD command and receive corresponding parameters.

**Note:** Commands sent by this function are short, so they are sent using polling transactions. The function does not return before the command transfer is completed. If any queued transactions sent by
esp_lcd_panel_io_tx_color() are still pending when this function is called, this function will wait until they are finished and the queue is empty before sending the command(s).

### Parameters
- **io** [in] LCD panel IO handle, which is created by other factory API like esp_lcd_new_panel_io_spi()
- **lcd_cmd** [in] The specific LCD command, set to -1 if no command needed
- **param** [out] Buffer for the command data
- **param_size** [in] Size of param buffer

### Returns
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

```c
esp_err_t esp_lcd_panel_io_tx_color(esp_lcd_panel_io_handle_t io, int lcd_cmd, const void *color, size_t color_size)
```

Transmit LCD RGB data.

### Note:
This function will package the command and RGB data into a transaction, and push into a queue. The real transmission is performed in the background (DMA+interrupt). The caller should take care of the lifecycle of the color buffer. Recycling of color buffer should be done in the callback on_color_trans_done().

```c
esp_err_t esp_lcd_panel_io_tx_color(esp_lcd_panel_io_handle_t io, int lcd_cmd, const void *color, size_t color_size)
```
**esp_err_t esp_lcd_panel_io_del (esp_lcd_panel_io_handle_t io)**

Destroy LCD panel IO handle (deinitialize panel and free all corresponding resource)

**Parameters**
- `io` [in] LCD panel IO handle, which is created by factory API like `esp_lcd_new_panel_io_spi()`

**Returns**
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

**esp_err_t esp_lcd_panel_io_register_event_callbacks (esp_lcd_panel_io_handle_t io, const esp_lcd_panel_io_callbacks_t *cbs, void *user_ctx)**

Register LCD panel IO callbacks.

**Parameters**
- `io` [in] LCD panel IO handle, which is created by factory API like `esp_lcd_new_panel_io_spi()`
- `cbs` [in] structure with all LCD panel IO callbacks
- `user_ctx` [in] User private data, passed directly to callback’s user_ctx

**Returns**
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

**esp_err_t esp_lcd_new_panel_io_spi (esp_lcd_spi_bus_handle_t bus, const esp_lcd_panel_io_spi_config_t *io_config, esp_lcd_panel_io_handle_t *ret_io)**

Create LCD panel IO handle, for SPI interface.

**Parameters**
- `bus` [in] SPI bus handle
- `io_config` [in] IO configuration, for SPI interface
- `ret_io` [out] Returned IO handle

**Returns**
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

**esp_err_t esp_lcd_new_panel_io_i2c_v1 (uint32_t bus, const esp_lcd_panel_io_i2c_config_t *io_config, esp_lcd_panel_io_handle_t *ret_io)**

Create LCD panel IO handle, for I2C interface in legacy implementation.

**Note:** Please don’t call this function in your project directly. Please call `esp_lcd_new_panel_to_i2c` instead.

**Parameters**
- `bus` [in] I2C bus handle, (in uint32_t)
- `io_config` [in] IO configuration, for I2C interface
- `ret_io` [out] Returned IO handle

**Returns**
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

**esp_err_t esp_lcd_new_panel_io_i2c_v2 (i2c_master_bus_handle_t bus, const esp_lcd_panel_io_i2c_config_t *io_config, esp_lcd_panel_io_handle_t *ret_io)**

Create LCD panel IO handle, for I2C interface in new implementation.
**Note:** Please don’t call this function in your project directly. Please call `esp_lcd_new_panel_to_i2c` instead.

### Parameters
- **bus** — [in] I2C bus handle, (in i2c_master_dev_handle_t)
- **io_config** — [in] IO configuration, for I2C interface
- **ret_io** — [out] Returned IO handle

### Returns
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

```c
esp_err_t esp_lcd_new_i80_bus(const esp_lcd_i80_bus_config_t *bus_config, esp_lcd_i80_bus_handle_t *ret_bus)
```

Create Intel 8080 bus handle.

### Parameters
- **bus_config** — [in] Bus configuration
- **ret_bus** — [out] Returned bus handle

### Returns
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_NOT_FOUND if no free bus is available
- ESP_OK on success

```c
esp_err_t esp_lcd_del_i80_bus(esp_lcd_i80_bus_handle_t bus)
```

Destroy Intel 8080 bus handle.

### Parameters
- **bus** — [in] Intel 8080 bus handle, created by `esp_lcd_new_i80_bus()`

### Returns
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if there still be some device attached to the bus
- ESP_OK on success

```c
esp_err_t esp_lcd_new_panel_io_i80(esp_lcd_i80_bus_handle_t bus, const esp_lcd_panel_io_i80_config_t *io_config, esp_lcd_panel_io_handle_t *ret_io)
```

Create LCD panel IO, for Intel 8080 interface.

### Parameters
- **bus** — [in] Intel 8080 bus handle, created by `esp_lcd_new_i80_bus()`
- **io_config** — [in] IO configuration, for i80 interface
- **ret_io** — [out] Returned panel IO handle

### Returns
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_SUPPORTED if some configuration can’t be satisfied, e.g. pixel clock out of the range
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

### Structures

```c
struct esp_lcd_panel_io_event_data_t
Type of LCD panel IO event data.
```

```c
struct esp_lcd_panel_io_callbacks_t
Type of LCD panel IO callbacks.
```
Public Members

`esp_lcd_panel_io_color_trans_done_cb_t on_color_trans_done`
Callback invoked when color data transfer has finished

`struct esp_lcd_panel_io_spi_config_t`
Panel IO configuration structure, for SPI interface.

Public Members

`int cs_gpio_num`
GPIO used for CS line

`int dc_gpio_num`
GPIO used to select the D/C line, set this to -1 if the D/C line is not used

`int spi_mode`
Traditional SPI mode (0~3)

`unsigned int pclk_hz`
Frequency of pixel clock

`size_t trans_queue_depth`
Size of internal transaction queue

`esp_lcd_panel_io_color_trans_done_cb_t on_color_trans_done`
Callback invoked when color data transfer has finished

`void *user_ctx`
User private data, passed directly to on_color_trans_done’s user_ctx

`int lcd_cmd_bits`
Bit-width of LCD command

`int lcd_param_bits`
Bit-width of LCD parameter

`unsigned int dc_low_on_data`
If this flag is enabled, DC line = 0 means transfer data, DC line = 1 means transfer command; vice versa

`unsigned int octal_mode`
Transmit with octal mode (8 data lines), this mode is used to simulate Intel 8080 timing

`unsigned int quad_mode`
Transmit with quad mode (4 data lines), this mode is useful when transmitting LCD parameters (Only use one line for command)
unsigned int **sio_mode**
   Read and write through a single data line (MOSI)

unsigned int **lsb_first**
   transmit LSB bit first

unsigned int **cs_high_active**
   CS line is high active

struct **esp_lcd_panel_io_spi_config_t**::[anonymous] flags
   Extra flags to fine-tune the SPI device

struct **esp_lcd_panel_io_i2c_config_t**
   Panel IO configuration structure, for I2C interface.

**Public Members**

uint32_t **dev_addr**
   I2C device address

**esp_lcd_panel_io_color_trans_done_cb_t** on_color_trans_done
   Callback invoked when color data transfer has finished

void **user_ctx**
   User private data, passed directly to on_color_trans_done’s user_ctx

size_t **control_phase_bytes**
   I2C LCD panel will encode control information (e.g. D/C selection) into control phase, in several bytes

unsigned int **dc_bit_offset**
   Offset of the D/C selection bit in control phase

int **lcd_cmd_bits**
   Bit-width of LCD command

int **lcd_param_bits**
   Bit-width of LCD parameter

unsigned int **dc_low_on_data**
   If this flag is enabled, DC line = 0 means transfer data, DC line = 1 means transfer command; vice versa

unsigned int **disable_control_phase**
   If this flag is enabled, the control phase isn’t used

struct **esp_lcd_panel_io_i2c_config_t**::[anonymous] flags
   Extra flags to fine-tune the I2C device
uint32_t scl_speed_hz
I2C LCD SCL frequency (hz)

struct esp_lcd_i80_bus_config_t
LCD Intel 8080 bus configuration structure.

Public Members

int dc_gpio_num
GPIO used for D/C line

int wr_gpio_num
GPIO used for WR line

lcd_clock_source_t clk_src
Clock source for the 80 LCD peripheral

int data_gpio_num[24]
GPiOs used for data lines

size_t bus_width
Number of data lines, 8 or 16

size_t max_transfer_bytes
Maximum transfer size, this determines the length of internal DMA link

size_t psram_trans_align
DMA transfer alignment for data allocated from PSRAM

size_t sram_trans_align
DMA transfer alignment for data allocated from SRAM

struct esp_lcd_panel_io_i80_config_t
Panel IO configuration structure, for intel 8080 interface.

Public Members

int cs_gpio_num
GPIO used for CS line, set to -1 will declaim exclusively use of 80 bus

uint32_t pclk_hz
Frequency of pixel clock

size_t trans_queue_depth
Transaction queue size, larger queue, higher throughput
esp_lcd_panel_io_color_trans_done_cb_t on_color_trans_done
Callback invoked when color data was transferred done

void *user_ctx
User private data, passed directly to on_color_trans_done’s user_ctx

int lcd_cmd_bits
Bit-width of LCD command

int lcd_param_bits
Bit-width of LCD parameter

unsigned int dc_idle_level
Level of DC line in IDLE phase

unsigned int dc_cmd_level
Level of DC line in CMD phase

unsigned int dc_dummy_level
Level of DC line in DUMMY phase

unsigned int dc_data_level
Level of DC line in DATA phase

struct esp_lcd_panel_io_i80_config_t::[anonymous] dc_levels
Each i80 device might have its own D/C control logic

unsigned int cs_active_high
If set, a high level of CS line will select the device, otherwise, CS line is low level active

unsigned int reverse_color_bits
Reverse the data bits, D[1:N] -> D[0:N]

unsigned int swap_color_bytes
Swap adjacent two color bytes

unsigned int pclk_active_neg
The display will write data lines when there’s a falling edge on WR signal (a.k.a the PCLK)

unsigned int pclk_idle_low
The WR signal (a.k.a the PCLK) stays at low level in IDLE phase

struct esp_lcd_panel_io_i80_config_t::[anonymous] flags
Panel IO config flags

Macros
esp_lcd_new_panel_io_i2c (bus, io_config, ret_io)
Create LCD panel IO handle.
Chapter 2. API Reference

Parameters
- **bus** [in] I2C bus handle
- **io_config** [in] IO configuration, for I2C interface
- **ret_io** [out] Returned IO handle

Returns
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

Type Definitions

typedef void *esp_lcd_spi_bus_handle_t
Type of LCD SPI bus handle

typedef uint32_t esp_lcd_i2c_bus_handle_t
Type of LCD I2C bus handle

typedef struct esp_lcd_i80_bus_t *esp_lcd_i80_bus_handle_t
Type of LCD intel 8080 bus handle

typedef bool (*esp_lcd_panel_io_color_trans_done_cb_t)(esp_lcd_panel_io_handle_t panel_io, esp_lcd_panel_io_event_data_t *edata, void *user_ctx)
Declare the prototype of the function that will be invoked when panel IO finishes transferring color data.

Param panel_io [in] LCD panel IO handle, which is created by factory API like esp_lcd_new_panel_io_spi()
Param edata [in] Panel IO event data, fed by driver
Param user_ctx [in] User data, passed from esp_lcd_panel_io_xxx_config_t
Return Whether a high priority task has been waken up by this function

Header File

- components/esp_lcd/include/esp_lcd_panel_ops.h
- This header file can be included with:

```c
#include "esp_lcd_panel_ops.h"
```
- This header file is a part of the API provided by the esp_lcd component. To declare that your component depends on esp_lcd, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_lcd
```

or

```cmake
PRIV_REQUIRES esp_lcd
```

Functions

*esp_err_t* **esp_lcd_panel_reset** *(esp_lcd_panel_handle_t panel)*

Reset LCD panel.

**Note:** Panel reset must be called before attempting to initialize the panel using esp_lcd_panel_init().

Parameters **panel** [in] LCD panel handle, which is created by other factory API like esp_lcd_new_panel_st7789()

Returns
- ESP_OK on success
**esp_err_t** esp_lcd_panel_init(esp_lcd_panel_handle_t panel)

Initialize LCD panel.

**Note:** Before calling this function, make sure the LCD panel has finished the reset stage by esp_lcd_panel_reset().

**Parameters**

- **panel** [in] LCD panel handle, which is created by other factory API like esp_lcd_new_panel_st7789()

**Returns**

- ESP_OK on success

**esp_err_t** esp_lcd_panel_del(esp_lcd_panel_handle_t panel)

Deinitialize the LCD panel.

**Parameters**

- **panel** [in] LCD panel handle, which is created by other factory API like esp_lcd_new_panel_st7789()

**Returns**

- ESP_OK on success

**esp_err_t** esp_lcd_panel_draw_bitmap(esp_lcd_panel_handle_t panel, int x_start, int y_start, int x_end, int y_end, const void* color_data)

Draw bitmap on LCD panel.

**Parameters**

- **panel** [in] LCD panel handle, which is created by other factory API like esp_lcd_new_panel_st7789()
- **x_start** [in] Start index on x-axis (x_start included)
- **y_start** [in] Start index on y-axis (y_start included)
- **x_end** [in] End index on x-axis (x_end not included)
- **y_end** [in] End index on y-axis (y_end not included)
- **color_data** [in] RGB color data that will be dumped to the specific window range

**Returns**

- ESP_OK on success

**esp_err_t** esp_lcd_panel_mirror(esp_lcd_panel_handle_t panel, bool mirror_x, bool mirror_y)

Mirror the LCD panel on specific axis.

**Note:** Combined with esp_lcd_panel_swap_xy(), one can realize screen rotation

**Parameters**

- **panel** [in] LCD panel handle, which is created by other factory API like esp_lcd_new_panel_st7789()
- **mirror_x** [in] Whether the panel will be mirrored about the x axis
- **mirror_y** [in] Whether the panel will be mirrored about the y axis

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

**esp_err_t** esp_lcd_panel_swap_xy(esp_lcd_panel_handle_t panel, bool swap_axes)

Swap/Exchange x and y axis.

**Note:** Combined with esp_lcd_panel_mirror(), one can realize screen rotation

**Parameters**
• **panel** - [in] LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
• **swap_axes** - [in] Whether to swap the x and y axis

**Returns**
- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

```c
esp_err_t esp_lcd_panel_set_gap(esp_lcd_panel_handle_t panel, int x_gap, int y_gap)
```
Set extra gap in x and y axis.
The gap is the space (in pixels) between the left/top sides of the LCD panel and the first row/column respectively of the actual contents displayed.

**Note:** Setting a gap is useful when positioning or centering a frame that is smaller than the LCD.

**Parameters**
- **panel** - [in] LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **x_gap** - [in] Extra gap on x axis, in pixels
- **y_gap** - [in] Extra gap on y axis, in pixels

**Returns**
- ESP_OK on success

```c
esp_err_t esp_lcd_panel_invert_color(esp_lcd_panel_handle_t panel, bool invert_color_data)
```
Invert the color (bit-wise invert the color data line)

**Parameters**
- **panel** - [in] LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **invert_color_data** - [in] Whether to invert the color data

**Returns**
- ESP_OK on success

```c
esp_err_t esp_lcd_panel_disp_on_off(esp_lcd_panel_handle_t panel, bool on_off)
```
Turn on or off the display.

**Parameters**
- **panel** - [in] LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **on_off** - [in] True to turns on display, False to turns off display

**Returns**
- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

```c
esp_err_t esp_lcd_panel_disp_off(esp_lcd_panel_handle_t panel, bool off)
```
Turn off the display.

**Parameters**
- **panel** - [in] LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()`
- **off** - [in] Whether to turn off the screen

**Returns**
- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

```c
esp_err_t esp_lcd_panel_disp_sleep(esp_lcd_panel_handle_t panel, bool sleep)
```
Enter or exit sleep mode.

**Parameters**
• **panel** -[in] LCD panel handle, which is created by other factory API like `esp_lcd_new_panel_st7789()
• **sleep** -[in] True to enter sleep mode, False to wake up

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if this function is not supported by the panel

**Header File**

- `components/esp_lcd/include/esp_lcd_panel_rgb.h`
- This header file can be included with:

  ```
  #include "esp_lcd_panel_rgb.h"
  ```

- This header file is a part of the API provided by the esp_lcd component. To declare that your component depends on esp_lcd, add the following to your CMakeLists.txt:

  ```
  REQUIRES esp_lcd
  ```
  or

  ```
  PRIV_REQUIRES esp_lcd
  ```

- `components/esp_lcd/include/esp_lcd_panel_vendor.h`
- This header file can be included with:

  ```
  #include "esp_lcd_panel_vendor.h"
  ```

- This header file is a part of the API provided by the esp_lcd component. To declare that your component depends on esp_lcd, add the following to your CMakeLists.txt:

  ```
  REQUIRES esp_lcd
  ```
  or

  ```
  PRIV_REQUIRES esp_lcd
  ```

**2.5.14 LED Control (LEDC)**

**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. It has 8 channels which can generate independent waveforms that can be used, for example, to drive RGB LED devices.

The PWM controller can automatically increase or decrease the duty cycle gradually, allowing for fades without any processor interference.

**Functionality Overview**

Setting up a channel of the LEDC is done in three steps. Note that unlike ESP32, ESP32-S2 only supports configuring channels in “low speed” mode.

1. **Timer Configuration** by specifying the PWM signal’s frequency and duty cycle resolution.
2. **Channel Configuration** 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 fade end.

![Fig. 13: Key Settings of LED PWM Controller’s API](image)

**Note:** For an initial setup, it is recommended to configure for the timers first (by calling `ledc_timer_config()`), and then for the channels (by calling `ledc_channel_config()`). This ensures the PWM frequency is at the desired value since the appearance of the PWM signal from the IO pad.

**Timer Configuration** Setting the timer is done by calling the function `ledc_timer_config()` and passing the data structure `ledc_timer_config_t` that contains the following configuration settings:

- Speed mode (value must be `LEDC_LOW_SPEED_MODE`)
- Timer number `ledc_timer_t`
- PWM signal frequency in Hz
- Resolution of PWM duty
- Source clock `ledc_clk_cfg_t`

The frequency and the duty resolution are interdependent. The higher the PWM frequency, the lower the duty resolution which 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**.

The source clock can also limit the PWM frequency. The higher the source clock frequency, the higher the maximum PWM frequency can be configured.

<table>
<thead>
<tr>
<th>Clock name</th>
<th>Clock freq</th>
<th>Clock capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>APB_CLK</td>
<td>80 MHz</td>
<td>/</td>
</tr>
<tr>
<td>REF_TICK</td>
<td>1 MHz</td>
<td>Dynamic Frequency Scaling compatible</td>
</tr>
<tr>
<td>RC_FAST_CLK</td>
<td>~ 8 MHz</td>
<td>Dynamic Frequency Scaling compatible, Light sleep compatible</td>
</tr>
<tr>
<td>XTAL_CLK</td>
<td>40 MHz</td>
<td>Dynamic Frequency Scaling compatible</td>
</tr>
</tbody>
</table>
Chapter 2. API Reference

Note:

1. On ESP32-S2, if RC_FAST_CLK is chosen as the LEDC clock source, an internal calibration will be performed to get the exact frequency of the clock. This ensures the accuracy of output PWM signal frequency.

The LEDC driver offers a helper function `ledc_find_suitable_duty_resolution()` to find the maximum possible resolution for the timer, given the source clock frequency and the desired PWM signal frequency.

When a timer is no longer needed by any channel, it can be deconfigured by calling the same function `ledc_timer_config()`. The configuration structure `ledc_timer_config_t` passes in should be:

- `ledc_timer_config_t::speed_mode` The speed mode of the timer which wants to be deconfigured belongs to `ledc_mode_t`.
- `ledc_timer_config_t::timer_num` The ID of the timers which wants to be deconfigured `ledc_timer_t`.
- `ledc_timer_config_t::deconfigure` Set this to true so that the timer specified can be deconfigured.

Channel Configuration  When the timer is set up, configure the desired 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.

Note:  All the timers and channels in the ESP32-S2’s LED PWM Controller only support low speed mode. Any change of PWM settings must be explicitly triggered by software (see below).

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 Channel Configuration.

The range of the duty cycle values passed to functions depends on selected `duty_resolution` and should be from 0 to \((2^{**\text{duty_resolution}})\). For example, if the selected duty resolution is 10, then the duty cycle values can range from 0 to 1024. This provides the resolution of \(\sim 0.1\%\).

Warning:  On ESP32-S2, when channel’s binded timer selects its maximum duty resolution, the duty cycle value cannot be set to \((2^{**\text{duty_resolution}})\). Otherwise, the internal duty counter in the hardware will overflow and be messed up.

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.
Chapter 2. API Reference

- `ledc_set_fade_with_time()`
- `ledc_set_fade_with_step()`
- `ledc_set_fade()`

Start fading with `ledc_fade_start()`. A fade can be operated in blocking or non-blocking mode, please check `ledc_fade_mode_t` for the difference between the two available fade modes. Note that with either fade mode, the next fade or fixed-duty update will not take effect until the last fade finishes or is stopped. `ledc_fade_stop()` has to be called to stop a fade that is in progress.

To get a notification about the completion of a fade operation, a fade end callback function can be registered for each channel by calling `ledc_cb_register()` after the fade service being installed. The fade end callback prototype is defined in `ledc_cb_t`, where you should return a boolean value from the callback function, indicating whether a high priority task is woken up by this callback function. It is worth mentioning, the callback and the function invoked by itself should be placed in IRAM, as the interrupt service routine is in IRAM. `ledc_cb_register()` will print a warning message if it finds the addresses of callback and user context are incorrect.

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 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()`.

**Supported Range of Frequency and Duty Resolutions**  

The LED PWM Controller is designed primarily to drive LEDs. It provides a large flexibility of PWM duty cycle settings. For instance, the PWM frequency of 5 kHz can have the maximum duty resolution of 13 bits. This 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). Note, however, that these parameters depend on the clock signal clocking the LED PWM Controller timer which in turn clocks the channel (see `timer configuration` and the ESP32-S2 Technical Reference Manual > LED PWM Controller (LEDC) [PDF]).

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 results in the following error reported on a serial monitor:
In such a situation, either the duty resolution or the frequency must be reduced. For example, setting the duty resolution to 2 resolves this issue and makes it possible to set the duty cycle at 25% steps, i.e., at 25%, 50% or 75%.

The LEDC driver also captures and reports 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 reducing freq_hz or duty_resolution. div_param=128
```

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 basic example: `peripherals/ledc/ledc_basic`.

The LEDC change duty cycle and fading control example: `peripherals/ledc/ledc_fade`.

**API Reference**

**Header File**

- `components/driver/ledc/include/driver/ledc.h`
- This header file can be included with:
  ```
  #include "driver/ledc.h"
  ```
- This header file is a part of the API provided by the `driver` component. To declare that your component depends on `driver`, add the following to your `CMakeLists.txt`:
  ```
  REQUIRES driver
  ```
  or
  ```
  PRIV_REQUIRES driver
  ```

**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.

**Parameters**

- `ledc_conf` - Pointer of LEDC channel configure struct

**Returns**

- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

`uint32_t ledc_find_suitable_duty_resolution` (uint32_t `src_clk_freq`, uint32_t `timer_freq`)

Helper function to find the maximum possible duty resolution in bits for ledc_channel_config()

**Parameters**

- `src_clk_freq` - LEDC timer source clock frequency (Hz) (See doxygen comments of `ledc_clk_cfg_t` or get from `esp_clk_tree_src_get_freq_hz`)
- `timer_freq` - Desired LEDC timer frequency (Hz)

**Returns**

- 0 The timer frequency cannot be achieved
- Others The largest duty resolution value to be set
esp_err_t ledc_timer_config (const ledc_timer_config_t *timer_conf)

LEDCC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution.

Parameters timer_conf – Pointer of LEDC timer configure struct

Returns
  • 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.
  • ESP_ERR_INVALID_STATE Timer cannot be de-configured because timer is not configured or is not paused

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

Note: If CONFIG_LEDC_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, makes it possible to execute even when the Cache is disabled.

Note: This function is allowed to run within ISR context.

Parameters
  • speed_mode – Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
  • channel – LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns
  • ESP_OK Success
  • ESP_ERR_INVALID_ARG Parameter error

esp_err_t ledc_set_pin (int gpio_num, ledc_mode_t speed_mode, ledc_channel_t ledc_channel)

Set LEDC output gpio.

Note: This function only routes the LEDC signal to GPIO through matrix, other LEDC resources initialization are not involved. Please use ledc_channel_config() instead to fully configure a LEDC channel.

Parameters
  • gpio_num – The LEDC output gpio
  • speed_mode – Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
  • ledc_channel – LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns
  • ESP_OK Success
  • ESP_ERR_INVALID_ARG Parameter error
**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.

**Note:** If CONFIG_LEDC_CTRL_FUNC_IN_IRAM is enabled, this function will be placed in the IRAM by linker, making it possible to execute even when the Cache is disabled.

**Note:** This function is allowed to run within ISR context.

**Parameters**
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **idle_level** - Set output idle level after LEDC stops.

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

**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)

**Parameters**
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_num** - LEDC timer index (0-3), select from ledc_timer_t
- **freq_hz** - Set the LEDC frequency

**Returns**
- 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.

**uint32_t ledc_get_freq** (ledc_mode_t speed_mode, ledc_timer_t timer_num)

LEDC get channel frequency (Hz)

**Parameters**
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_num** - LEDC timer index (0-3), select from ledc_timer_t

**Returns**
- 0 error
- Others Current LEDC frequency

**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:** For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.
Chapter 2. API Reference

Parameters
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **duty** - Set the LEDC duty, the range of duty setting is \([0, (2^{n \text{duty resolution}})]\)
- **hpoint** - Set the LEDC hpoint value, the range is \([0, (2^{n \text{duty resolution}})-1]\)

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

```c
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.

Parameters
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns
- LEDC_ERR_VAL if parameter error
- Others Current hpoint value of LEDC channel

```c
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: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

```c
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.

Parameters
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t

Returns
- ESP_OK Success
- ESP_ERR_INVALIDARGV Parameter error

```c
uint32_t ledc_get_duty (ledc_mode_t speed_mode, ledc_channel_t channel)
```
• Others Current LEDC duty

```c
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:** For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

**Parameters**
- `speed_mode` - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- `channel` - LEDC channel (0 - LEDC_CHANNEL_MAX-1), 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

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

```c
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.

**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 (ORed) 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.

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_NOT_FOUND Failed to find available interrupt source

```c
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.

**Parameters**
- `speed_mode` - Select the LEDC channel group with specified speed mode. Note that not all targets support high 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 is [1, SOC_LEDC_TIMER_BIT_WIDTH]
- `clk_src` - Select LEDC source clock.

**Returns**
- (-1) Parameter error
- Other Current LEDC duty
**esp_err_t ledc_timer_rst**(ledc_mode_t speed_mode, ledc_timer_t timer_sel)

Reset LEDC timer.

**Parameters**
- **speed_mode** – Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** – LEDC timer index (0-3), select from ledc_timer_t

**Returns**
- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

**esp_err_t ledc_timer_pause**(ledc_mode_t speed_mode, ledc_timer_t timer_sel)

Pause LEDC timer counter.

**Parameters**
- **speed_mode** – Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** – LEDC timer index (0-3), select from ledc_timer_t

**Returns**
- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

**esp_err_t ledc_timer_resume**(ledc_mode_t speed_mode, ledc_timer_t timer_sel)

Resume LEDC timer.

**Parameters**
- **speed_mode** – Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **timer_sel** – LEDC timer index (0-3), select from ledc_timer_t

**Returns**
- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

**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.

**Parameters**
- **speed_mode** – Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** – LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **timer_sel** – LEDC timer index (0-3), select from ledc_timer_t

**Returns**
- ESP_ERR_INVALID_ARG Parameter error
- ESP_OK Success

**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_setFade_with_step, ledc_setFade_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: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **target_duty** - Target duty of fading [0, (2**duty_resolution)]
- **scale** - Controls the increase or decrease step scale.
- **cycle_num** - Increase or decrease the duty every cycle_num cycles

Returns
- ESP_OK Success
- ESP.ERR_INVALID_ARG Parameter error
- ESP.ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

`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: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

Parameters
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **target_duty** - Target duty of fading [0, (2**duty_resolution)]
- **max_fade_time_ms** - The maximum time of the fading (ms).

Returns
- ESP_OK Success
- ESP.ERR_INVALID_ARG Parameter error
- ESP.ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

`esp_err_t ledc_fade_func_install (int intr_alloc_flags)`

Install LEDC fade function. This function will occupy interrupt of LEDC module.

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.

Returns
- ESP_OK Success
- ESP.ERR_INVALID_ARG Intr flag error
• ESP_ERR_NOT_FOUND Failed to find available interrupt source
• ESP_ERR_INVALID_STATE Fade function already installed

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:** Starting fade operation with this API is not thread-safe, use with care.

**Note:** For ESP32, hardware does not support any duty change while a fade operation is running in progress
on that channel. Other duty operations will have to wait until the fade operation has finished.

**Parameters**
- **speed_mode** – Select the LEDC channel group with specified speed mode. Note that
not all targets support high 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.

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_STATE Channel not initialized or fade function not installed.
- ESP_ERR_INVALID_ARG Parameter error.

esp_err_t ledc_fade_stop (ledc_mode_t speed_mode, ledc_channel_t channel)
Stop LEDC fading. The duty of the channel is guaranteed to be fixed at most one PWM cycle after the function
returns.

**Note:** This API can be called if a new fixed duty or a new fade want to be set while the last fade operation is
still running in progress.

**Note:** Call this API will abort the fading operation only if it was started by calling ledc_fade_start with
LEDC_FADE_NO_WAIT mode.

**Note:** If a fade was started with LEDC_FADE_WAIT_DONE mode, calling this API afterwards has no use
in stopping the fade. Fade will continue until it reaches the target duty.

**Parameters**
- **speed_mode** – Select the LEDC channel group with specified speed mode. Note that
  not all targets support high speed mode.
- **channel** – LEDC channel number

**Returns**
- ESP_OK Success
Chapter 2. API Reference

- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Fade function init error

```c
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:** For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

**Parameters**
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **duty** - Set the LEDC duty, the range of duty setting is \([0, (2^{n\text{duty_resolution}})]\)
- **hpoint** - Set the LEDC hpoint value, the range is \([0, (2^{n\text{duty_resolution}})-1]\)

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_ERR_INVALID_ARG Parameter error
- ESP_FAIL Fade function init error

```c
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:** For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

**Parameters**
- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from ledc_channel_t
- **target_duty** - Target duty of fading \([0, (2^{n\text{duty_resolution}})]\)
- **max_fade_time_ms** - The maximum time of the fading (ms).
- **fade_mode** - Choose blocking or non-blocking mode

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error
- ESP_ERR_INVALID_STATE Channel not initialized
- ESP_FAIL Fade function init error

```c
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: For ESP32, hardware does not support any duty change while a fade operation is running in progress on that channel. Other duty operations will have to wait until the fade operation has finished.

### Parameters

- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from `ledc_channel_t`
- **target_duty** - Target duty of fading [0, (2**duty_resolution)]
- **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

### Returns

- ESP_OK - Success
- ESP_ERR_INVALID_ARG - Parameter error
- ESP_ERR_INVALID_STATE - Channel not initialized
- ESP_FAIL - Fade function init error

```c
esp_err_t ledc_cb_register (ledc_mode_t speed_mode, ledc_channel_t channel, ledc_cbs_t *cbs, void *user_arg)
```

LEDC callback registration function.

Note: The callback is called from an ISR, it must never attempt to block, and any FreeRTOS API called must be ISR capable.

### Parameters

- **speed_mode** - Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- **channel** - LEDC channel index (0 - LEDC_CHANNEL_MAX-1), select from `ledc_channel_t`
- **cbs** - Group of LEDC callback functions
- **user_arg** - User registered data for the callback function

### Returns

- ESP_OK - Success
- ESP_ERR_INVALID_ARG - Parameter error
- ESP_ERR_INVALID_STATE - Channel not initialized
- ESP_FAIL - Fade function init error

### Structures

```c
struct ledc_channel_config_t
```

Configuration parameters of LEDC channel for `ledc_channel_config` function.

### Public Members

```c
int gpio_num
```

the LEDC output gpio_num, if you want to use gpio16, gpio_num = 16
**Chapter 2. API Reference**

**ledc_mode_t speed_mode**
LEDCC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode

**ledc_channel_t channel**
LEDCC channel (0 - LEDC_CHANNEL_MAX-1)

**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 - LEDC_TIMER_MAX-1)

uint32_t duty
LEDCC channel duty, the range of duty setting is [0, (2**duty_resolution)]

int hpoint
LEDCC channel hpoint value, the range is [0, (2**duty_resolution)-1]

unsigned int output_invert
Enable (1) or disable (0) gpio output invert

struct ledc_channel_config_t::[anonymous] flags
LEDCC flags

struct ledc_timer_config_t
Configuration parameters of LEDCC timer for ledc_timer_config function.

**Public Members**

**ledc_mode_t speed_mode**
LEDCC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode

**ledc_timer_bit_t duty_resolution**
LEDCC channel duty resolution

**ledc_timer_t timer_num**
The timer source of channel (0 - LEDC_TIMER_MAX-1)

uint32_t freq_hz
LEDCC timer frequency (Hz)

**ledc_clk_cfg_t clk_cfg**
Configure LEDCC source clock from ledc_clk_cfg_t. Note that LEDC_USE_RC_FAST_CLK and LEDC_USE_XTAL_CLK are non-timer-specific clock sources. You can not have one LEDC timer uses RC_FAST_CLK as the clock source and have another LEDC timer uses XTAL_CLK as its clock source. All chips except esp32 and esp32s2 do not have timer-specific clock sources, which means clock source for all timers must be the same one.
bool deconfigure

Set this field to de-configure a LEDC timer which has been configured before. Note that it will not check
whether the timer wants to be de-configured is binded to any channel. Also, the timer has to be paused
first before it can be de-configured. When this field is set, duty_resolution, freq_hz, clk_cfg fields are
ignored.

struct ledc_cb_param_t

LEDC callback parameter.

**Public Members**

`ledek_event_t event`

Event name

`uint32_t speed_mode`

Speed mode of the LEDC channel group

`uint32_t channel`

LEDC channel (0 - LEDC_CHANNEL_MAX-1)

`uint32_t duty`

LEDC current duty of the channel, the range of duty is [0, \(2^{\text{duty_resolution}}\)]

struct ledc_cbs_t

Group of supported LEDC callbacks.

**Note:** The callbacks are all running under ISR environment

**Public Members**

`ledek_t fade_cb`

LEDC fade_end callback function

**Macros**

`LEDC_APB_CLK_HZ`

Frequency of one of the LEDC peripheral clock sources, APB_CLK.

**Note:** This macro should have no use in your application, we keep it here only for backward compatible

`LEDC_REF_CLK_HZ`

Frequency of one of the LEDC peripheral clock sources, REF_TICK.

**Note:** This macro should have no use in your application, we keep it here only for backward compatible
LED_C_ERR_DUTY

LED_C_ERR_VAL

Type Definitions
typedef intr_handle_t ledc_isr_handle_t
typedef bool (*ledc_cb_t)(const ledc_cb_param_t *param, void *user_arg)

Type of LEDC event callback.

Param param LEDC callback parameter
Param user_arg User registered data
Return Whether a high priority task has been waken up by this function

Enumerations
enum ledc_cb_event_t

LEDC callback event type.

Values:

enumerator LEDC_FADE_END_EVT

LEDC fade end event

Header File

• components/hal/include/hal/ledc_types.h
• This header file can be included with:

#include "hal/ledc_types.h"

Type Definitions
typedef soc_periph_ledc_clk_src_legacy_t ledc_clk_cfg_t

LEDC clock source configuration struct.

In theory, the following enumeration shall be placed in LEDC driver’s header. However, as the next enumeration, ledc_clk_src_t, makes the use of some of these values and to avoid mutual inclusion of the headers, we must define it here.

Enumerations
enum ledc_mode_t

Values:

enumerator LEDC_LOW_SPEED_MODE

LEDC low speed speed_mode

enumerator LEDC_SPEED_MODE_MAX

LEDC speed limit
enum `ledc_intr_type_t`

Values:

- enumerator `LEDC_INTR_DISABLE`
  - Disable LEDC interrupt

- enumerator `LEDC_INTR_FADE_END`
  - Enable LEDC interrupt

- enumerator `LEDC_INTR_MAX`

enum `ledc_duty_direction_t`

Values:

- enumerator `LEDC_DUTY_DIR_DECREASE`
  - LEDC duty decrease direction

- enumerator `LEDC_DUTY_DIR_INCREASE`
  - LEDC duty increase direction

- enumerator `LEDC_DUTY_DIR_MAX`

enum `ledc_slow_clk_sel_t`

LED C global clock sources.

Values:

- enumerator `LEDC_SLOW_CLK_RC_FAST`
  - LEDC low speed timer clock source is RC_FAST clock

- enumerator `LEDC_SLOW_CLK_APB`
  - LEDC low speed timer clock source is 80MHz APB clock

- enumerator `LEDC_SLOW_CLK_XTAL`
  - LEDC low speed timer clock source XTAL clock

- enumerator `LEDC_SLOW_CLK_RTC8M`
  - Alias of ‘LEDC_SLOW_CLK_RC_FAST’

enum `ledc_clk_src_t`

LED C timer-specific clock sources.

Note: Setting numeric values to match `ledc_clk_cfg_t` values are a hack to avoid collision with `LEDC_AUTO_CLK` in the driver, as these enums have very similar names and user may pass one of these by mistake.

Values:

- enumerator `LEDC_REF_TICK`
  - LEDC timer clock divided from reference tick (1Mhz)
enumerator **LEDC_APB_CLK**
   LEDC timer clock divided from APB clock (80Mhz)

enumerator **LEDC_SCLK**
   Selecting this value for LEDC_TICK_SEL_TIMER let the hardware take its source clock from LEDC_APB_CLK_SEL.

**enum ledc_timer_t**

   **Values:**

   enumerator **LEDC_TIMER_0**
      LEDC timer 0

   enumerator **LEDC_TIMER_1**
      LEDC timer 1

   enumerator **LEDC_TIMER_2**
      LEDC timer 2

   enumerator **LEDC_TIMER_3**
      LEDC timer 3

   enumerator **LEDC_TIMER_MAX**

**enum ledc_channel_t**

   **Values:**

   enumerator **LEDC_CHANNEL_0**
      LEDC channel 0

   enumerator **LEDC_CHANNEL_1**
      LEDC channel 1

   enumerator **LEDC_CHANNEL_2**
      LEDC channel 2

   enumerator **LEDC_CHANNEL_3**
      LEDC channel 3

   enumerator **LEDC_CHANNEL_4**
      LEDC channel 4

   enumerator **LEDC_CHANNEL_5**
      LEDC channel 5

   enumerator **LEDC_CHANNEL_6**
      LEDC channel 6
enumerator LEDC_CHANNEL_7
    LEDC channel 7

enumerator LEDC_CHANNEL_MAX

enum ledc_timer_bit_t
Values:

enumerator LEDC_TIMER_1_BIT
    LEDC PWM duty resolution of 1 bits

enumerator LEDC_TIMER_2_BIT
    LEDC PWM duty resolution of 2 bits

enumerator LEDC_TIMER_3_BIT
    LEDC PWM duty resolution of 3 bits

enumerator LEDC_TIMER_4_BIT
    LEDC PWM duty resolution of 4 bits

enumerator LEDC_TIMER_5_BIT
    LEDC PWM duty resolution of 5 bits

enumerator LEDC_TIMER_6_BIT
    LEDC PWM duty resolution of 6 bits

enumerator LEDC_TIMER_7_BIT
    LEDC PWM duty resolution of 7 bits

enumerator LEDC_TIMER_8_BIT
    LEDC PWM duty resolution of 8 bits

enumerator LEDC_TIMER_9_BIT
    LEDC PWM duty resolution of 9 bits

enumerator LEDC_TIMER_10_BIT
    LEDC PWM duty resolution of 10 bits

enumerator LEDC_TIMER_11_BIT
    LEDC PWM duty resolution of 11 bits

enumerator LEDC_TIMER_12_BIT
    LEDC PWM duty resolution of 12 bits

enumerator LEDC_TIMER_13_BIT
    LEDC PWM duty resolution of 13 bits

enumerator LEDC_TIMER_14_BIT
    LEDC PWM duty resolution of 14 bits
Chapter 2. API Reference

enumerator **LEDC_TIMER_BIT_MAX**

elem **ledc_fade_mode_t**

   Values:

   enumerator **LEDC_FADE_NO_WAIT**
      LEDC fade function will return immediately

   enumerator **LEDC_FADE_WAIT DONE**
      LEDC fade function will block until fading to the target duty

   enumerator **LEDC_FADE_MAX**

2.5.15 Pulse Counter (PCNT)

Introduction

The PCNT (Pulse Counter) module is designed to count the number of rising and/or falling edges of input signals. The ESP32-S2 contains multiple pulse counter units in the module. Each unit is in effect an independent counter with multiple channels, where each channel can increment/decrement the counter on a rising/falling edge. Furthermore, each channel can be configured separately.

PCNT channels can react to signals of edge type and level type, however for simple applications, detecting the edge signal is usually sufficient. PCNT channels can be configured react to both pulse edges (i.e., rising and falling edge), and can be configured to increase, decrease or do nothing to the unit’s counter on each edge. The level signal is the so-called control signal, which is used to control the counting mode of the edge signals that are attached to the same channel. By combining the usage of both edge and level signals, a PCNT unit can act as a quadrature decoder.

Besides that, PCNT unit is equipped with a separate glitch filter, which is helpful to remove noise from the signal.

Typically, a PCNT module can be used in scenarios like:

   - Calculate periodic signal’s frequency by counting the pulse numbers within a time slice
   - Decode quadrature signals into speed and direction

Functional Overview

Description of the PCNT functionality is divided into the following sections:

- **Resource Allocation** - covers how to allocate PCNT units and channels with properly set of configurations. It also covers how to recycle the resources when they finished working.
- **Set Up Channel Actions** - covers how to configure the PCNT channel to behave on different signal edges and levels.
- **Watch Points** - describes how to configure PCNT watch points (i.e., tell PCNT unit to trigger an event when the count reaches a certain value).
- **Register Event Callbacks** - describes how to hook your specific code to the watch point event callback function.
- **Set Glitch Filter** - describes how to enable and set the timing parameters for the internal glitch filter.
- **Enable and Disable Unit** - describes how to enable and disable the PCNT unit.
- **Unit IO Control** - describes IO control functions of PCNT unit, like enable glitch filter, start and stop unit, get and clear count value.
- **Power Management** - describes what functionality will prevent the chip from going into low power mode.

---

1 Different ESP chip series might have different number of PCNT units and channels. Please refer to the [TRM] for details. The driver does not forbid you from applying for more PCNT units and channels, but it returns error when all available hardware resources are used up. Please always check the return value when doing resource allocation (e.g., `pcnt_new_unit()`).
Chapter 2. API Reference

- **IRAM Safe** - describes tips on how to make the PCNT interrupt and IO control functions work better along with a disabled cache.
- **Thread Safety** - lists which APIs are guaranteed to be thread safe by the driver.
- **Kconfig Options** - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

**Resource Allocation** The PCNT unit and channel are represented by `pcnt_unit_handle_t` and `pcnt_channel_handle_t` respectively. All available units and channels are maintained by the driver in a resource pool, so you do not need to know the exact underlying instance ID.

**Install PCNT Unit** To install a PCNT unit, there is a configuration structure that needs to be given in advance:

```c
pcnt_unit_config_t
```

- `pcnt_unit_config_t::low_limit` and `pcnt_unit_config_t::high_limit` specify the range for the internal hardware counter. The counter will reset to zero automatically when it crosses either the high or low limit.
- `pcnt_unit_config_t::accum_count` sets whether to create an internal accumulator for the counter. This is helpful when you want to extend the counter’s width, which by default is 16 bit at most, defined in the hardware. See also **Compensate Overflow Loss** for how to use this feature to compensate the overflow loss.
- `pcnt_unit_config_t::intr_priority` sets the priority of the interrupt. If it is set to 0, the driver will allocate an interrupt with a default priority. Otherwise, the driver will use the given priority.

**Note:** Since all PCNT units share the same interrupt source, when installing multiple PCNT units make sure that the interrupt priority `pcnt_unit_config_t::intr_priority` is the same for each unit.

Unit allocation and initialization is done by calling a function `pcnt_new_unit()` with `pcnt_unit_config_t` as an input parameter. The function will return a PCNT unit handle only when it runs correctly. Specifically, when there are no more free PCNT units in the pool (i.e., unit resources have been used up), then this function will return `ESP_ERR_NOT_FOUND` error. The total number of available PCNT units is recorded by `SOC_PCNT_UNITS_PER_GROUP` for reference.

If a previously created PCNT unit is no longer needed, it is recommended to recycle the resource by calling `pcnt_del_unit()`. Which in return allows the underlying unit hardware to be used for other purposes. Before deleting a PCNT unit, one should ensure the following prerequisites:

- The unit is in the init state, in other words, the unit is either disabled by `pcnt_unit_disable()` or not enabled yet.
- The attached PCNT channels are all removed by `pcnt_del_channel()`.

```
#define EXAMPLE_PCNT_HIGH_LIMIT 100
#define EXAMPLE_PCNT_LOW_LIMIT -100

pcnt_unit_config_t unit_config = {
  .high_limit = EXAMPLE_PCNT_HIGH_LIMIT,
  .low_limit = EXAMPLE_PCNT_LOW_LIMIT,
};

pcnt_unit_handle_t pcnt_unit = NULL;
ESP_ERROR_CHECK(pcnt_new_unit(&unit_config, &pcnt_unit));
```

**Install PCNT Channel** To install a PCNT channel, you must initialize a `pcnt_chan_config_t` structure in advance, and then call `pcnt_new_channel()`. The configuration fields of the `pcnt_chan_config_t` structure are described below:

- `pcnt_chan_config_t::edge_gpio_num` and `pcnt_chan_config_t::level_gpio_num` specify the GPIO numbers used by **edge** type signal and **level** type signal. Please note, either of them can be assigned to -1 if it is not actually used, and thus it will become a virtual IO. For some simple pulse counting applications where one of the level/edge signals is fixed (i.e., never changes), you can reclaim a GPIO
by setting the signal as a virtual IO on channel allocation. Setting the level/edge signal as a virtual IO causes that signal to be internally routed to a fixed High/Low logic level, thus allowing you to save a GPIO for other purposes.

- `pcnt_chan_config_t::virt_edge_io_level` and `pcnt_chan_config_t::virt_level_io_level` specify the virtual IO level for `edge` and `level` input signal, to ensure a deterministic state for such control signal. Please note, they are only valid when either `pcnt_chan_config_t::edge_gpio_num` or `pcnt_chan_config_t::level_gpio_num` is assigned to -1.
- `pcnt_chan_config_t::invert_edge_input` and `pcnt_chan_config_t::invert_level_input` are used to decide whether to invert the input signals before they going into PCNT hardware. The invert is done by GPIO matrix instead of PCNT hardware.
- `pcnt_chan_config_t::io_loop_back` is for debug only, which enables both the GPIO’s input and output paths. This can help to simulate the pulse signals by function `gpio_set_level()` on the same GPIO.

Channel allocating and initialization is done by calling a function `pcnt_new_channel()` with the above `pcnt_chan_config_t` as an input parameter plus a PCNT unit handle returned from `pcnt_new_unit()`. This function will return a PCNT channel handle if it runs correctly. Specifically, when there are no more free PCNT channel within the unit (i.e., channel resources have been used up), then this function will return `ESP_ERR_NOT_FOUND` error. The total number of available PCNT channels within the unit is recorded by `SOC_PCNT_CHANNELS_PER_UNIT` for reference. Note that, when install a PCNT channel for a specific unit, one should ensure the unit is in the init state, otherwise this function will return `ESP_ERR_INVALID_STATE` error.

If a previously created PCNT channel is no longer needed, it is recommended to recycle the resources by calling `pcnt_del_channel()`. Which in return allows the underlying channel hardware to be used for other purposes.

```c
#define EXAMPLE_CHAN_GPIO_A 0
#define EXAMPLE_CHAN_GPIO_B 2

pcnt_chan_config_t chan_config = {
  .edge_gpio_num = EXAMPLE_CHAN_GPIO_A,
  .level_gpio_num = EXAMPLE_CHAN_GPIO_B,
};
pcnt_channel_handle_t pcnt_chan = NULL;
ESP_ERROR_CHECK(pcnt_new_channel(pcnt_unit, &chan_config, &pcnt_chan));

Set Up Channel Actions  The PCNT will increase/decrease/hold its internal count value when the input pulse signal toggles. You can set different actions for edge signal and/or level signal.

- `pcnt_channel_set_edge_action()` function is to set specific actions for rising and falling edge of the signal attached to the `pcnt_chan_config_t::edge_gpio_num`. Supported actions are listed in `pcnt_channel_edge_action_t`.
- `pcnt_channel_set_level_action()` function is to set specific actions for high and low level of the signal attached to the `pcnt_chan_config_t::level_gpio_num`. Supported actions are listed in `pcnt_channel_level_action_t`. This function is not mandatory if the `pcnt_chan_config_t::level_gpio_num` is set to -1 when allocating PCNT channel by `pcnt_new_channel()`.

```c
// decrease the counter on rising edge, increase the counter on falling edge
ESP_ERROR_CHECK(pcnt_channel_set_edge_action(pcnt_chan, PCNT_CHANNEL_EDGE_ACTION_DECREASE, PCNT_CHANNEL_EDGE_ACTION_INCREASE));
// keep the counting mode when the control signal is high level, and reverse the counting mode when the control signal is low level
ESP_ERROR_CHECK(pcnt_channel_set_level_action(pcnt_chan, PCNT_CHANNEL_LEVEL_ACTION_KEEP, PCNT_CHANNEL_LEVEL_ACTION_INVERSE));
```

Watch Points  Each PCNT unit can be configured to watch several different values that you are interested in. The value to be watched is also called **Watch Point**. The watch point itself can not exceed the range set in `pcnt_unit_config_t` by `pcnt_unit_config_t::low_limit` and
pcnt_unit_config_t::high_limit. When the counter reaches either watch point, a watch event will be triggered and notify you by interrupt if any watch event callback has ever registered in `pcnt_unit_register_event_callbacks()`. See `Register Event Callbacks` for how to register event callbacks.

The watch point can be added and removed by `pcnt_unit_add_watch_point()` and `pcnt_unit_remove_watch_point()`. The commonly-used watch points are: `zero cross`, `maximum/minimum count` and other threshold values. The number of available watch point is limited, `pcnt_unit_add_watch_point()` will return error `ESP_ERR_NOT_FOUND` if it can not find any free hardware resource to save the watch point. You can not add the same watch point for multiple times, otherwise it will return error `ESP_ERR_INVALID_STATE`.

It is recommended to remove the unused watch point by `pcnt_unit_remove_watch_point()` to recycle the watch point resources.

```c
// add zero across watch point
ESP_ERROR_CHECK(pcnt_unit_add_watch_point(pcnt_unit, 0));
// add high limit watch point
ESP_ERROR_CHECK(pcnt_unit_add_watch_point(pcnt_unit, EXAMPLE_PCNT_HIGH_LIMIT));
```

**Note:** Due to the hardware limitation, after adding a watch point, you should call `pcnt_unit_clear_count()` to make it take effect.

**Register Event Callbacks** When PCNT unit reaches any enabled watch point, specific event will be generated and notify the CPU by interrupt. If you have some function that want to get executed when event happens, you should hook your function to the interrupt service routine by calling `pcnt_unit_register_event_callbacks()`.

All supported event callbacks are listed in the `pcnt_event_callbacks_t`:

- `pcnt_event_callbacks_t::on_reach` sets a callback function for watch point event. As this function is called within the ISR context, you must ensure that the function does not attempt to block (e.g., by making sure that only FreeRTOS APIs with `ISR` suffix are called from within the function). The function prototype is declared in `pcnt_watch_cb_t`.

You can save their own context to `pcnt_unit_register_event_callbacks()` as well, via the parameter `user_ctx`. This user data will be directly passed to the callback functions.

In the callback function, the driver will fill in the event data of specific event. For example, the watch point event data is declared as `pcnt_watch_event_data_t`:

- `pcnt_watch_event_data_t::watch_point_value` saves the watch point value that triggers the event.
- `pcnt_watch_event_data_t::zero_cross_mode` saves how the PCNT unit crosses the zero point in the latest time. The possible zero cross modes are listed in the `pcnt_unit_zero_cross_mode_t`. Usually different zero cross mode means different counting direction and counting step size.

Registering callback function results in lazy installation of interrupt service, thus this function should only be called before the unit is enabled by `pcnt_unit_enable()`. Otherwise, it can return `ESP_ERR_INVALID_STATE` error.

```c
static bool example_pcnt_on_reach(pcnt_unit_handle_t unit, const pcnt_watch_event_data_t *edata, void *user_ctx)
{
  BaseType_t high_task_wakeup;
  QueueHandle_t queue = (QueueHandle_t)user_ctx;
  // send watch point to queue, from this interrupt callback
  xQueueSendFromISR(queue, &edata->watch_point_value, &high_task_wakeup);
  // return whether a high priority task has been waken up by this function
  return (high_task_wakeup == pdTRUE);
}
```
pcnt_event_callbacks_t cbs = {
    .on_reach = example_pcnt_on_reach,
};
QueueHandle_t queue = xQueueCreate(10, sizeof(int));
ESP_ERROR_CHECK(pcnt_unit_register_event_callbacks(pcnt_unit, &cbs, queue));

Set Glitch Filter  The PCNT unit features filters to ignore possible short glitches in the signals. The parameters that can be configured for the glitch filter are listed in `pcnt_glitch_filter_config_t`:

- `pcnt_glitch_filter_config_t::max_glitch_ns` sets the maximum glitch width, in nanoseconds. If a signal pulse’s width is smaller than this value, then it will be treated as noise and will not increase/decrease the internal counter.

You can enable the glitch filter for PCNT unit by calling `pcnt_unit_set_glitch_filter()` with the filter configuration provided above. Particularly, you can disable the glitch filter later by calling `pcnt_unit_set_glitch_filter()` with a NULL filter configuration.

This function should be called when the unit is in the init state. Otherwise, it will return `ESP_ERR_INVALID_STATE` error.

Note:  The glitch filter is clocked from APB. For the counter not to miss any pulses, the maximum glitch width should be longer than one APB_CLK cycle (usually 12.5 ns if APB equals 80 MHz). As the APB frequency would be changed after DFS (Dynamic Frequency Scaling) enabled, which means the filter does not work as expect in that case. So the driver installs a PM lock for PCNT unit during the first time you enable the glitch filter. For more information related to power management strategy used in PCNT driver, please see `Power Management`.

pcnt_glitch_filter_config_t filter_config = {
    .max_glitch_ns = 1000,
};
ESP_ERROR_CHECK(pcnt_unit_set_glitch_filter(pcnt_unit, &filter_config));

Enable and Disable Unit  Before doing IO control to the PCNT unit, you need to enable it first, by calling `pcnt_unit_enable()`. Internally, this function:

- switches the PCNT driver state from init to enable.
- enables the interrupt service if it has been lazy installed in `pcnt_unit_register_event_callbacks()`.
- acquires a proper power management lock if it has been lazy installed in `pcnt_unit_set_glitch_filter()`. See also `Power Management` for more information.

On the contrary, calling `pcnt_unit_disable()` will do the opposite, that is, put the PCNT driver back to the init state, disable the interrupts service and release the power management lock.

Unit IO Control

Start/Stop and Clear  Calling `pcnt_unit_start()` makes the PCNT unit start to work, increase or decrease counter according to pulse signals. On the contrary, calling `pcnt_unit_stop()` will stop the PCNT unit but retain current count value. Instead, clearing counter can only be done by calling `pcnt_unit_clear_count()`.

Note, `pcnt_unit_start()` and `pcnt_unit_stop()` should be called when the unit has been enabled by `pcnt_unit_enable()`. Otherwise, it will return `ESP_ERR_INVALID_STATE` error.

Get Count Value  You can read current count value at any time by calling `pcnt_unit_get_count()`. The returned count value is a `signed` integer, where the sign can be used to reflect the direction.
```c
int pulse_count = 0;
ESP_ERROR_CHECK(pcnt_unit_get_count(pcnt_unit, &pulse_count));
```

**Compensate Overflow Loss** The internal hardware counter will be cleared to zero automatically when it reaches high or low limit. If you want to compensate for that count loss and extend the counter’s bit-width, you can:

1. Enable `pcnt_unit_config_t::accum_count` when installing the PCNT unit.
2. Add the high/low limit as the Watch Points.
3. Now, the returned count value from the `pcnt_unit_get_count()` function not only reflects the hardware’s count value, but also accumulates the high/low overflow loss to it.

**Note:** `pcnt_unit_clear_count()` resets the accumulated count value as well.

**Power Management** When power management is enabled (i.e., `CONFIG_PM_ENABLE` is on), the system will adjust the APB frequency before going into light sleep, thus potentially changing the behavior of PCNT glitch filter and leading to valid signal being treated as noise.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type `ESP_PM_APB_FREQ_MAX`. Whenever you enable the glitch filter by `pcnt_unit_set_glitch_filter()`, the driver guarantees that the power management lock is acquired after the PCNT unit is enabled by `pcnt_unit_enable()`. Likewise, the driver releases the lock after `pcnt_unit_disable()` is called.

**IRAM Safe** By default, the PCNT interrupt will be deferred when the Cache is disabled for reasons like writing/erasing Flash. Thus the alarm interrupt will not get executed in time, which is not expected in a real-time application.

There is a Kconfig option `CONFIG_PCNT_ISR_IRAM_SAFE` that:

1. Enables the interrupt being serviced even when cache is disabled
2. Places all functions that used by the ISR into IRAM
3. Places driver object into DRAM (in case it is mapped to PSRAM by accident)

This allows the interrupt to run while the cache is disabled but comes at the cost of increased IRAM consumption.

There is another Kconfig option `CONFIG_PCNT_CTRL_FUNC_IN_IRAM` that can put commonly used IO control functions into IRAM as well. So that these functions can also be executable when the cache is disabled. These IO control functions are as follows:

- `pcnt_unit_start()`
- `pcnt_unit_stop()`
- `pcnt_unit_clear_count()`
- `pcnt_unit_get_count()`

**Thread Safety** The factory functions `pcnt_new_unit()` and `pcnt_new_channel()` are guaranteed to be thread safe by the driver, which means, you can call them from different RTOS tasks without protection by extra locks.

The following functions are allowed to run under ISR context, the driver uses a critical section to prevent them being called concurrently in both task and ISR.

- `pcnt_unit_start()`
- `pcnt_unit_stop()`
- `pcnt_unit_clear_count()`
- `pcnt_unit_get_count()`

---

2 `pcnt_event_callbacks_t::on_reach` callback and the functions invoked by itself should also be placed in IRAM, you need to take care of them by themselves.
Other functions that take the `pcnt_unit_handle_t` and `pcnt_channel_handle_t` as the first positional parameter, are not treated as thread safe. This means you should avoid calling them from multiple tasks.

**Kconfig Options**

- `CONFIG_PCNT_CTRL_FUNC_IN_IRAM` controls where to place the PCNT control functions (IRAM or Flash), see IRAM Safe for more information.
- `CONFIG_PCNT_ISR_IRAM_SAFE` controls whether the default ISR handler can work when cache is disabled, see IRAM Safe for more information.
- `CONFIG_PCNT_ENABLE_DEBUG_LOG` is used to enabled the debug log output. Enabling this option increases the firmware binary size.

**Application Examples**

- Decode the quadrature signals from rotary encoder: peripherals/pcnt/rotary_encoder.

**API Reference**

**Header File**

- `components/esp_driver_pcnt/include/driver/pulse_cnt.h`

  This header file can be included with:

  ```c
  #include "driver/pulse_cnt.h"
  ```

  This header file is a part of the API provided by the esp_driver_pcnt component. To declare that your component depends on esp_driver_pcnt, add the following to your CMakeLists.txt:

  ```cmake
  REQUIRES esp_driver_pcnt
  ```

  or

  ```cmake
  PRIV_REQUIRES esp_driver_pcnt
  ```

**Functions**

`esp_err_t pcnt_new_unit (const pcnt_unit_config_t *config, pcnt_unit_handle_t *ret_unit)`

Create a new PCNT unit, and return the handle.

**Note:** The newly created PCNT unit is put in the init state.

**Parameters**

- `config` [in] PCNT unit configuration
- `ret_unit` [out] Returned PCNT unit handle

**Returns**

- ESP_OK: Create PCNT unit successfully
- ESP_ERR_INVALID_ARG: Create PCNT unit failed because of invalid argument (e.g. high/low limit value out of the range)
- ESP_ERR_NO_MEM: Create PCNT unit failed because out of memory
- ESP_ERR_NOT_FOUND: Create PCNT unit failed because all PCNT units are used up and no more free one
- ESP_FAIL: Create PCNT unit failed because of other error

`esp_err_t pcnt_del_unit (pcnt_unit_handle_t unit)`

Delete the PCNT unit handle.
**Note:** A PCNT unit can’t be in the enable state when this function is invoked. See also `pcnt_unit_disable()` for how to disable a unit.

**Parameters**
- `unit` - [in] PCNT unit handle created by `pcnt_new_unit()`

**Returns**
- ESP_OK: Delete the PCNT unit successfully
- ESP_ERR_INVALID_ARG: Delete the PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Delete the PCNT unit failed because the unit is not in init state or some PCNT channel is still in working
- ESP_FAIL: Delete the PCNT unit failed because of other error

```c
esp_err_t pcnt_unit_set_glitch_filter( pcnt_unit_handle_t unit, const pcnt_glitch_filter_config_t *config )
```

Set glitch filter for PCNT unit.

**Note:** The glitch filter module is clocked from APB, and APB frequency can be changed during DFS, which in return make the filter out of action. So this function will lazy-install a PM lock internally when the power management is enabled. With this lock, the APB frequency won’t be changed. The PM lock can be uninstalled in `pcnt_del_unit()`.

**Note:** This function should be called when the PCNT unit is in the init state (i.e. before calling `pcnt_unit_enable()`)

**Parameters**
- `unit` - [in] PCNT unit handle created by `pcnt_new_unit()`
- `config` - [in] PCNT filter configuration, set config to NULL means disabling the filter function

**Returns**
- ESP_OK: Set glitch filter successfully
- ESP_ERR_INVALID_ARG: Set glitch filter failed because of invalid argument (e.g. glitch width is too big)
- ESP_ERR_INVALID_STATE: Set glitch filter failed because the unit is not in the init state
- ESP_FAIL: Set glitch filter failed because of other error

```c
esp_err_t pcnt_unit_enable( pcnt_unit_handle_t unit )
```

Enable the PCNT unit.

**Note:** This function will transit the unit state from init to enable.

**Note:** This function will enable the interrupt service, if it’s lazy installed in `pcnt_unit_register_event_callbacks()`.

**Note:** This function will acquire the PM lock if it’s lazy installed in `pcnt_unit_set_glitch_filter()`.

**Note:** Enable a PCNT unit doesn’t mean to start it. See also `pcnt_unit_start()` for how to start the
PCNT counter.

**Parameters**  
unit – [in] PCNT unit handle created by `pcnt_new_unit()`

**Returns**
- ESP_OK: Enable PCNT unit successfully
- ESP_ERR_INVALID_ARG: Enable PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable PCNT unit failed because the unit is already enabled
- ESP_FAIL: Enable PCNT unit failed because of other error

```c
esp_err_t pcnt_unit_disable(pcnt_unit_handle_t unit)
```

Disable the PCNT unit.

**Note:** This function will do the opposite work to the `pcnt_unit_enable()`

**Note:** Disable a PCNT unit doesn’t mean to stop it. See also `pcnt_unit_stop()` for how to stop the PCNT counter.

**Parameters**  
unit – [in] PCNT unit handle created by `pcnt_new_unit()`

**Returns**
- ESP_OK: Disable PCNT unit successfully
- ESP_ERR_INVALID_ARG: Disable PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable PCNT unit failed because the unit is not enabled yet
- ESP_FAIL: Disable PCNT unit failed because of other error

```c
esp_err_t pcnt_unit_start(pcnt_unit_handle_t unit)
```

Start the PCNT unit, the counter will start to count according to the edge and/or level input signals.

**Note:** This function should be called when the unit is in the enable state (i.e. after calling `pcnt_unit_enable()`)

**Note:** This function is allowed to run within ISR context

**Note:** This function will be placed into IRAM if `CONFIG_PCNT_CTRL_FUNC_IN_IRAM` is on, so that it’s allowed to be executed when Cache is disabled

**Parameters**  
unit – [in] PCNT unit handle created by `pcnt_new_unit()`

**Returns**
- ESP_OK: Start PCNT unit successfully
- ESP_ERR_INVALID_ARG: Start PCNT unit failed because of invalid argument
- ESP_ERR_INVALID_STATE: Start PCNT unit failed because the unit is not enabled yet
- ESP_FAIL: Start PCNT unit failed because of other error

```c
esp_err_t pcnt_unit_stop(pcnt_unit_handle_t unit)
```

Stop PCNT from counting.
**Note:** This function should be called when the unit is in the enable state (i.e. after calling \texttt{pcnt\_unit\_enable()}).

**Note:** The stop operation won’t clear the counter. Also see \texttt{pcnt\_unit\_clear\_count()} for how to clear pulse count value.

**Note:** This function is allowed to run within ISR context

**Note:** This function will be placed into IRAM if \texttt{CONFIG\_PCNT\_CTRL\_FUNC\_IN\_IRAM}, so that it is allowed to be executed when Cache is disabled

### \texttt{pcnt\_unit\_clear\_count (pcnt\_unit\_handle\_t unit)}

Clear PCNT pulse count value to zero.

**Note:** It’s recommended to call this function after adding a watch point by \texttt{pcnt\_unit\_add\_watch\_point()}, so that the newly added watch point is effective immediately.

**Note:** This function is allowed to run within ISR context

**Note:** This function will be placed into IRAM if \texttt{CONFIG\_PCNT\_CTRL\_FUNC\_IN\_IRAM}, so that it’s allowed to be executed when Cache is disabled

### \texttt{pcnt\_unit\_get\_count (pcnt\_unit\_handle\_t unit, int*value)}

Get PCNT count value.

**Note:** This function is allowed to run within ISR context

**Note:** This function will be placed into IRAM if \texttt{CONFIG\_PCNT\_CTRL\_FUNC\_IN\_IRAM}, so that it’s allowed to be executed when Cache is disabled

### Parameters
• **unit** – [in] PCNT unit handle created by `pcnt_new_unit()`
• **value** – [out] Returned count value

**Returns**
• ESP_OK: Get PCNT pulse count successfully
• ESP_ERR_INVALID_ARG: Get PCNT pulse count failed because of invalid argument
• ESP_FAIL: Get PCNT pulse count failed because of other error

```c
esp_err_t pcnt_unit_register_event_callbacks(pcnt_unit_handle_t unit, const pcnt_event_callbacks_t *cbs, void *user_data)
```

Set event callbacks for PCNT unit.

**Note:** User registered callbacks are expected to be runnable within ISR context

**Note:** The first call to this function needs to be before the call to `pcnt_unit_enable`

**Note:** User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

**Parameters**
• **unit** – [in] PCNT unit handle created by `pcnt_new_unit()`
• **cbs** – [in] Group of callback functions
• **user_data** – [in] User data, which will be passed to callback functions directly

**Returns**
• ESP_OK: Set event callbacks successfully
• ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
• ESP_ERR_INVALID_STATE: Set event callbacks failed because the unit is not in init state
• ESP_FAIL: Set event callbacks failed because of other error

```c
esp_err_t pcnt_unit_add_watch_point(pcnt_unit_handle_t unit, int watch_point)
```

Add a watch point for PCNT unit, PCNT will generate an event when the counter value reaches the watch point value.

**Parameters**
• **unit** – [in] PCNT unit handle created by `pcnt_new_unit()`
• **watch_point** – [in] Value to be watched

**Returns**
• ESP_OK: Add watch point successfully
• ESP_ERR_INVALID_ARG: Add watch point failed because of invalid argument (e.g. the value to be watched is out of the limitation set in `pcnt_unit_config_t`)
• ESP_ERR_INVALID_STATE: Add watch point failed because the same watch point has already been added
• ESP_ERR_NOT_FOUND: Add watch point failed because no more hardware watch point can be configured
• ESP_FAIL: Add watch point failed because of other error

```c
esp_err_t pcnt_unit_remove_watch_point(pcnt_unit_handle_t unit, int watch_point)
```

Remove a watch point for PCNT unit.

**Parameters**
• **unit** – [in] PCNT unit handle created by `pcnt_new_unit()`
• **watch_point** – [in] Watch point value

**Returns**
• ESP_OK: Remove watch point successfully
Chapter 2. API Reference

• ESP_ERR_INVALID_ARG: Remove watch point failed because of invalid argument
• ESP_ERR_INVALID_STATE: Remove watch point failed because the watch point was not added by pcnt_unit_add_watch_point() yet
• ESP_FAIL: Remove watch point failed because of other error

\texttt{esp_err_t \textbf{pcnt_new_channel} (pcnt_unit_handle_t unit, const pcnt_chan_config_t *config, pcnt_channel_handle_t *ret_chan)}

Create PCNT channel for specific unit, each PCNT has several channels associated with it.

\textbf{Note:} This function should be called when the unit is in init state (i.e., before calling pcnt_unit_enable()).

\textbf{Parameters}

- \texttt{unit} - [in] PCNT unit handle created by pcnt_new_unit()
- \texttt{config} - [in] PCNT channel configuration
- \texttt{ret_chan} - [out] Returned channel handle

\textbf{Returns}

- ESP_OK: Create PCNT channel successfully
- ESP_ERR_INVALID_ARG: Create PCNT channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create PCNT channel failed because of insufficient memory
- ESP_ERR_NOT_FOUND: Create PCNT channel failed because all PCNT channels are used up and no more free one
- ESP_ERR_INVALID_STATE: Create PCNT channel failed because the unit is not in the init state
- ESP_FAIL: Create PCNT channel failed because of other error

\texttt{esp_err_t \textbf{pcnt_del_channel} (pcnt_channel_handle_t chan)}

Delete the PCNT channel.

\textbf{Parameters} \texttt{chan} - [in] PCNT channel handle created by pcnt_new_channel()

\textbf{Returns}

- ESP_OK: Delete the PCNT channel successfully
- ESP_ERR_INVALID_ARG: Delete the PCNT channel failed because of invalid argument
- ESP_FAIL: Delete the PCNT channel failed because of other error

\texttt{esp_err_t \textbf{pcnt_channel_set_edge_action} (pcnt_channel_handle_t chan, pcnt_channel_edge_action_t pos_act, pcnt_channel_edge_action_t neg_act)}

Set channel actions when edge signal changes (e.g., falling or rising edge occurred). The edge signal is input from the \texttt{edge_gpio_num} configured in pcnt_chan_config_t. We use these actions to control when and how to change the counter value.

\textbf{Parameters}

- \texttt{chan} - [in] PCNT channel handle created by pcnt_new_channel()
- \texttt{pos_act} - [in] Action on posedge signal
- \texttt{neg_act} - [in] Action on negedge signal

\textbf{Returns}

- ESP_OK: Set edge action for PCNT channel successfully
- ESP_ERR_INVALID_ARG: Set edge action for PCNT channel failed because of invalid argument
- ESP_FAIL: Set edge action for PCNT channel failed because of other error

\texttt{esp_err_t \textbf{pcnt_channel_set_level_action} (pcnt_channel_handle_t chan, pcnt_channel_level_action_t high_act, pcnt_channel_level_action_t low_act)}

Set channel actions when level signal changes (e.g., signal level goes from high to low). The level signal is input from the \texttt{level_gpio_num} configured in pcnt_chan_config_t. We use these actions to control when and how to change the counting mode.

\textbf{Parameters}
• chan - [in] PCNT channel handle created by `pcnt_new_channel()`
• high_act - [in] Action on high level signal
• low_act - [in] Action on low level signal

**Returns**
- ESP_OK: Set level action for PCNT channel successfully
- ESP_ERR_INVALID_ARG: Set level action for PCNT channel failed because of invalid argument
- ESP_FAIL: Set level action for PCNT channel failed because of other error

**Structures**

```c
struct pcnt_watch_event_data_t
```
PCNT watch event data.

**Public Members**

```c
int watch_point_value
```
Watch point value that triggered the event

```c
pcnt_unit_zero_cross_mode_t zero_cross_mode
```
Zero cross mode

```c
struct pcnt_event_callbacks_t
```
Group of supported PCNT callbacks.

**Note:** The callbacks are all running under ISR environment

**Note:** When `CONFIG_PCNT_ISR_IRAM_SAFE` is enabled, the callback itself and functions called by it should be placed in IRAM.

**Public Members**

```c
pcnt_watch_cb_t on_reach
```
Called when PCNT unit counter reaches any watch point

```c
struct pcnt_unit_config_t
```
PCNT unit configuration.

**Public Members**

```c
int low_limit
```
Low limitation of the count unit, should be lower than 0

```c
int high_limit
```
High limitation of the count unit, should be higher than 0
int `intr_priority`
   PCNT interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

`uint32_t accum_count`
   Whether to accumulate the count value when overflows at the high/low limit

`struct pcnt_unit_config_t`: [anonymous] `flags`
   Extra flags

`struct pcnt_chan_config_t`
   PCNT channel configuration.

**Public Members**

- `uint32_t edge_gpio_num`
  GPIO number used by the edge signal, input mode with pull up enabled. Set to -1 if unused

- `uint32_t level_gpio_num`
  GPIO number used by the level signal, input mode with pull up enabled. Set to -1 if unused

- `uint32_t invert_edge_input`
  Invert the input edge signal

- `uint32_t invert_level_input`
  Invert the input level signal

- `uint32_t virt_edge_io_level`
  Virtual edge IO level, 0: low, 1: high. Only valid when edge_gpio_num is set to -1

- `uint32_t virt_level_io_level`
  Virtual level IO level, 0: low, 1: high. Only valid when level_gpio_num is set to -1

- `uint32_t io_loop_back`
  For debug/test, the signal output from the GPIO will be fed to the input path as well

`struct pcnt_chan_config_t`: [anonymous] `flags`
   Channel config flags

`struct pcnt_glitch_filter_config_t`
   PCNT glitch filter configuration.

**Public Members**

- `uint32_t max_glitch_ns`
  Pulse width smaller than this threshold will be treated as glitch and ignored, in the unit of ns
Chapter 2. API Reference

Type Definitions

typedef struct pcnt_unit_t *pcnt_unit_handle_t
    Type of PCNT unit handle.

typedef struct pcnt_chan_t *pcnt_channel_handle_t
    Type of PCNT channel handle.

typedef bool (*pcnt_watch_cb_t)(pcnt_unit_handle_t unit, const pcnt_watch_event_data_t *edata, void *user_ctx)
    PCNT watch event callback prototype.

Note: The callback function is invoked from an ISR context, so it should meet the restrictions of not calling any blocking APIs when implementing the callback. E.g. must use ISR version of FreeRTOS APIs.

Param unit [in] PCNT unit handle
Param edata [in] PCNT event data, fed by the driver
Param user_ctx [in] User data, passed from pcnt_unit_register_event_callbacks()
Return Whether a high priority task has been woken up by this function

Header File

• components/hal/include/hal/pcnt_types.h
• This header file can be included with:

```c
#include "hal/pcnt_types.h"
```

Enumerations

enum pcnt_channel_level_action_t
    PCNT channel action on control level.
    Values:

    enumerator PCNT_CHANNEL_LEVEL_ACTION_KEEP
        Keep current count mode

    enumerator PCNT_CHANNEL_LEVEL_ACTION_INVERSE
        Invert current count mode (increase -> decrease, decrease -> increase)

    enumerator PCNT_CHANNEL_LEVEL_ACTION_HOLD
        Hold current count value

enum pcnt_channel_edge_action_t
    PCNT channel action on signal edge.
    Values:

    enumerator PCNT_CHANNEL_EDGE_ACTION_HOLD
        Hold current count value

    enumerator PCNT_CHANNEL_EDGE_ACTION_INCREASE
        Increase count value
enumerator PCNT_CHANNEL_EDGE_ACTION_DECREASE
Decrease count value

enum pcnt_unit_zero_cross_mode_t
PCNT unit zero cross mode.
Values:

denumerator PCNT_UNIT_ZERO_CROSS_POS_ZERO
start from positive value, end to zero, i.e. +N->0

denumerator PCNT_UNIT_ZERO_CROSS_NEG_ZERO
start from negative value, end to zero, i.e. -N->0

denumerator PCNT_UNIT_ZERO_CROSS_NEG_POS
start from negative value, end to positive value, i.e. -N->+M

denumerator PCNT_UNIT_ZERO_CROSS_POS_NEG
start from positive value, end to negative value, i.e. +N->-M

2.5.16 Remote Control Transceiver (RMT)

Introduction

The RMT (Remote Control Transceiver) peripheral was designed to act as an infrared transceiver. However, due to the flexibility of its data format, RMT can be extended to a versatile and general-purpose transceiver, transmitting or receiving many other types of signals. From the perspective of network layering, the RMT hardware contains both physical and data link layers. The physical layer defines the communication media and bit signal representation. The data link layer defines the format of an RMT frame. The minimal data unit in the frame is called the RMT symbol, which is represented by rmt_symbol_word_t in the driver.

ESP32-S2 contains multiple channels in the RMT peripheral\(^1\). Each channel can be independently configured as either transmitter or receiver.

Typically, the RMT peripheral can be used in the following scenarios:

- Transmit or receive infrared signals, with any IR protocols, e.g., NEC
- General-purpose sequence generator
- Transmit signals in a hardware-controlled loop, with a finite or infinite number of times
- Multi-channel simultaneous transmission
- Modulate the carrier to the output signal or demodulate the carrier from the input signal

Layout of RMT Symbols
The RMT hardware defines data in its own pattern—the RMT symbol. The diagram below illustrates the bit fields of an RMT symbol. Each symbol consists of two pairs of two values. The first value in the pair is a 15-bit value representing the signal’s duration in units of RMT ticks. The second in the pair is a 1-bit value representing the signal’s logic level, i.e., high or low.

RMT Transmitter Overview
The data path and control path of an RMT TX channel is illustrated in the figure below:

\(^1\) Different ESP chip series might have different numbers of RMT channels. Please refer to [TRM] for details. The driver does not forbid you from applying for more RMT channels, but it returns an error when there are no hardware resources available. Please always check the return value when doing Resource Allocation.
The driver encodes the user’s data into RMT data format, then the RMT transmitter can generate the waveforms according to the encoding artifacts. It is also possible to modulate a high-frequency carrier signal before being routed to a GPIO pad.

**RMT Receiver Overview**  
The data path and control path of an RMT RX channel is illustrated in the figure below:

The RMT receiver can sample incoming signals into RMT data format, and store the data in memory. It is also possible to tell the receiver the basic characteristics of the incoming signal, so that the signal’s stop condition can be recognized, and signal glitches and noise can be filtered out. The RMT peripheral also supports demodulating the high-frequency carrier from the base signal.

**Functional Overview**

The description of the RMT functionality is divided into the following sections:

- *Resource Allocation* - covers how to allocate and properly configure RMT channels. It also covers how to recycle channels and other resources when they are no longer used.
- *Carrier Modulation and Demodulation* - describes how to modulate and demodulate the carrier signals for TX and RX channels respectively.
- *Register Event Callbacks* - covers how to register user-provided event callbacks to receive RMT channel events.
- *Enable and Disable Channel* - shows how to enable and disable the RMT channel.
- *Initiate TX Transaction* - describes the steps to initiate a transaction for a TX channel.
- *Initiate RX Transaction* - describes the steps to initiate a transaction for an RX channel.
- *Multiple Channels Simultaneous Transmission* - describes how to collect multiple channels into a sync group so that their transmissions can be started simultaneously.
- *RMT Encoder* - focuses on how to write a customized encoder by combining multiple primitive encoders that are provided by the driver.
Chapter 2. API Reference

Fig. 16: RMT Receiver Overview

- **Power Management** - describes how different clock sources affect power consumption.
- **IRAM Safe** - describes how disabling the cache affects the RMT driver, and tips to mitigate it.
- **Thread Safety** - lists which APIs are guaranteed to be thread-safe by the driver.
- **Kconfig Options** - describes the various Kconfig options supported by the RMT driver.

**Resource Allocation**  Both RMT TX and RX channels are represented by `rmt_channel_handle_t` in the driver. The driver internally manages which channels are available and hands out a free channel on request.

**Install RMT TX Channel**  To install an RMT TX channel, there is a configuration structure that needs to be given in advance `rmt_tx_channel_config_t`. The following list describes each member of the configuration structure.

- `rmt_tx_channel_config_t::gpio_num` sets the GPIO number used by the transmitter.
- `rmt_tx_channel_config_t::clk_src` selects the source clock for the RMT channel. The available clocks are listed in `rmt_clock_source_t`. Note that, the selected clock is also used by other channels, which means the user should ensure this configuration is the same when allocating other channels, regardless of TX or RX. For the effect on the power consumption of different clock sources, please refer to the Power Management section.
- `rmt_tx_channel_config_t::resolution_hz` sets the resolution of the internal tick counter. The timing parameter of the RMT signal is calculated based on this tick.
- `rmt_tx_channel_config_t::mem_block_symbols` has a slightly different meaning based on if the DMA backend is enabled or not.
  - If the DMA is enabled via `rmt_tx_channel_config_t::with_dma`, then this field controls the size of the internal DMA buffer. To achieve a better throughput and smaller CPU overhead, you can set a larger value, e.g., 1024.
  - If DMA is not used, this field controls the size of the dedicated memory block owned by the channel, which should be at least 64.
- `rmt_tx_channel_config_t::trans_queue_depth` sets the depth of the internal transaction queue, the deeper the queue, the more transactions can be prepared in the backlog.
- `rmt_tx_channel_config_t::invert_out` is used to decide whether to invert the RMT signal before sending it to the GPIO pad.
- `rmt_tx_channel_config_t::with_dma` enables the DMA backend for the channel. Using the DMA allows a significant amount of the channel’s workload to be offloaded from the CPU. However, the DMA backend is not available on all ESP chips, please refer to [TRM] before you enable this option. Or you might encounter an ESP_ERR_NOT_SUPPORTED error.
- `rmt_tx_channel_config_t::io_loop_back` enables both input and output capabilities on the channel’s assigned GPIO. Thus, by binding a TX and RX channel to the same GPIO, loopback can be achieved.
- `rmt_tx_channel_config_t::io_od_mode` configures the channel’s assigned GPIO as open-drain. When combined with `rmt_tx_channel_config_t::io_loop_back`, a bi-directional bus (e.g., 1-
wire) can be achieved.

- `rmt_tx_channel_config_t::intr_priority` Set the priority of the interrupt. If set to 0, then the driver will use a interrupt with low or medium priority (priority level may be one of 1, 2 or 3), otherwise use the priority indicated by `rmt_tx_channel_config_t::intr_priority`. Please use the number form (1, 2, 3), not the bitmask form ((1<<1),(1<<2),(1<<3)). Please pay attention that once the interrupt priority is set, it cannot be changed until `rmt_del_channel()` is called.

Once the `rmt_tx_channel_config_t` structure is populated with mandatory parameters, users can call `rmt_new_tx_channel()` to allocate and initialize a TX channel. This function returns an RMT channel handle if it runs correctly. Specifically, when there are no more free channels in the RMT resource pool, this function returns `ESP_ERR_NOT_FOUND` error. If some feature (e.g., DMA backend) is not supported by the hardware, it returns `ESP_ERR_NOT_SUPPORTED` error.

```c
rmt_channel_handle_t tx_chan = NULL;
rmt_tx_channel_config_t tx_chan_config = {
  .clk_src = RMT_CLK_SRC_DEFAULT, // select source clock
  .gpio_num = 0, // GPIO number
  .mem_block_symbols = 64, // memory block size, 64 * 4 = 256 Bytes
  .resolution_hz = 1 * 1000 * 1000, // 1 MHz tick resolution, i.e., 1 tick = 1 µs
  .trans_queue_depth = 4, // set the number of transactions that can...← pend in the background
  .flags.invert_out = false, // do not invert output signal
  .flags.with_dma = false, // do not need DMA backend
};
ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_chan_config, &tx_chan));
```

**Install RMT RX Channel**

To install an RMT RX channel, there is a configuration structure that needs to be given in advance `rmt_rx_channel_config_t`. The following list describes each member of the configuration structure.

- `rmt_rx_channel_config_t::gpio_num` sets the GPIO number used by the receiver.
- `rmt_rx_channel_config_t::clk_src` selects the source clock for the RMT channel. The available clocks are listed in `rmt_clock_source_t`. Note that, the selected clock is also used by other channels, which means the user should ensure this configuration is the same when allocating other channels, regardless of TX or RX. For the effect on the power consumption of different clock sources, please refer to the Power Management section.
- `rmt_rx_channel_config_t::resolution_hz` sets the resolution of the internal tick counter. The timing parameter of the RMT signal is calculated based on this tick.
- `rmt_rx_channel_config_t::mem_block_symbols` has a slightly different meaning based on whether the DMA backend is enabled.
  - If the DMA is enabled via `rmt_rx_channel_config_t::with_dma`, this field controls the maximum size of the DMA buffer.
  - If DMA is not used, this field controls the size of the dedicated memory block owned by the channel, which should be at least 64.
- `rmt_rx_channel_config_t::invert_in` is used to invert the input signals before it is passed to the RMT receiver. The inversion is done by the GPIO matrix instead of by the RMT peripheral.
- `rmt_rx_channel_config_t::with_dma` enables the DMA backend for the channel. Using the DMA allows a significant amount of the channel’s workload to be offloaded from the CPU. However, the DMA backend is not available on all ESP chips, please refer to [TRM] before you enable this option. Or you might encounter a `ESP_ERR_NOT_SUPPORTED` error.
- `rmt_rx_channel_config_t::io_loop_back` enables both input and output capabilities on the channel’s assigned GPIO. Thus, by binding a TX and RX channel to the same GPIO, loopback can be achieved.
- `rmt_rx_channel_config_t::intr_priority` Set the priority of the interrupt. If set to 0, then the driver will use a interrupt with low or medium priority (priority level may be one of 1, 2 or 3), otherwise use the priority indicated by `rmt_rx_channel_config_t::intr_priority`. Please use the number form (1, 2, 3), not the bitmask form ((1<<1),(1<<2),(1<<3)). Please pay attention that once the interrupt priority is set, it cannot be changed until `rmt_del_channel()` is called.

Once the `rmt_rx_channel_config_t` structure is populated with mandatory parameters, users can call
rmt_new_rx_channel() to allocate and initialize an RX channel. This function returns an RMT channel handle if it runs correctly. Specifically, when there are no more free channels in the RMT resource pool, this function returns ESP_ERR_NOT_FOUND error. If some feature (e.g., DMA backend) is not supported by the hardware, it returns ESP_ERR_NOT_SUPPORTED error.

```c
rmt_channel_handle_t rx_chan = NULL;
rmt_rx_channel_config_t rx_chan_config = {
  .clk_src = RMT_CLK_SRC_DEFAULT, // select source clock
  .resolution_hz = 1 * 1000 * 1000, // 1 MHz tick resolution, i.e., 1 tick = 1 µs
  .mem_block_symbols = 64, // memory block size, 64 * 4 = 256 Bytes
  .gpio_num = 2, // GPIO number
  .flags.invert_in = false, // do not invert input signal
  .flags.with_dma = false, // do not need DMA backend
};
ESP_ERROR_CHECK(rmt_new_rx_channel(&rx_chan_config, &rx_chan));
```

**Note:** Due to a software limitation in the GPIO driver, when both TX and RX channels are bound to the same GPIO, ensure the RX Channel is initialized before the TX Channel. If the TX Channel was set up first, then during the RX Channel setup, the previous RMT TX Channel signal will be overridden by the GPIO control signal.

Uninstall RMT Channel If a previously installed RMT channel is no longer needed, it is recommended to recycle the resources by calling rmt_del_channel(), which in return allows the underlying software and hardware resources to be reused for other purposes.

Carrier Modulation and Demodulation The RMT transmitter can generate a carrier wave and modulate it onto the message signal. Compared to the message signal, the carrier signal’s frequency is significantly higher. In addition, the user can only set the frequency and duty cycle for the carrier signal. The RMT receiver can demodulate the carrier signal from the incoming signal. Note that, carrier modulation and demodulation are not supported on all ESP chips, please refer to [TRM] before configuring the carrier, or you might encounter a ESP_ERR_NOT_SUPPORTED error.

Carrier-related configurations lie in rmt_carrier_config_t:

- rmt_carrier_config_t::frequency_hz sets the carrier frequency, in Hz.
- rmt_carrier_config_t::duty_cycle sets the carrier duty cycle.
- rmt_carrier_config_t::polarity_active_low sets the carrier polarity, i.e., on which level the carrier is applied.
- rmt_carrier_config_t::always_on sets whether to output the carrier even when the data transmission has finished. This configuration is only valid for the TX channel.

**Note:** For the RX channel, we should not set the carrier frequency exactly to the theoretical value. It is recommended to leave a tolerance for the carrier frequency. For example, in the snippet below, we set the frequency to 25 KHz, instead of the 38 KHz configured on the TX side. The reason is that reflection and refraction occur when a signal travels through the air, leading to distortion on the receiver side.

```c
rmt_carrier_config_t tx_carrier_cfg = {
  .duty_cycle = 0.33, // duty cycle 33%
  .frequency_hz = 38000, // 38 KHz
  .flags.polarity_active_low = false, // carrier should be modulated to high_level
};
// modulate carrier to TX channel
ESP_ERROR_CHECK(rmt_apply_carrier(tx_chan, &tx_carrier_cfg));
```

```
```
Register Event Callbacks  When an event occurs on an RMT channel (e.g., transmission or receiving is completed), the CPU is notified of this event via an interrupt. If you have some function that needs to be called when a particular events occur, you can register a callback for that event to the RMT driver’s ISR (Interrupt Service Routine) by calling `rmt_tx_register_event_callbacks()` and `rmt_rx_register_event_callbacks()` for TX and RX channel respectively. Since the registered callback functions are called in the interrupt context, the user should ensure the callback function does not block, e.g., by making sure that only FreeRTOS APIs with the `FromISR` suffix are called from within the function. The callback function has a boolean return value used to indicate whether a higher priority task has been unblocked by the callback.

The TX channel-supported event callbacks are listed in the `rmt_tx_event_callbacks_t`:

- `rmt_tx_event_callbacks_t::on_trans_done` sets a callback function for the “trans-done” event. The function prototype is declared in `rmt_tx_done_callback_t`.

The RX channel-supported event callbacks are listed in the `rmt_rx_event_callbacks_t`:

- `rmt_rx_event_callbacks_t::on_recv_done` sets a callback function for “receive-done” event. The function prototype is declared in `rmt_rx_done_callback_t`.

Users can save their own context in `rmt_tx_register_event_callbacks()` and `rmt_rx_register_event_callbacks()` as well, via the parameter `user_data`. The user data is directly passed to each callback function.

In the callback function, users can fetch the event-specific data that is filled by the driver in the `edata`. Note that the `edata` pointer is only valid during the callback.

The TX-done event data is defined in `rmt_tx_done_event_data_t`:

- `rmt_tx_done_event_data_t::num_symbols` indicates the number of transmitted RMT symbols. This also reflects the size of the encoding artifacts. Please note, this value accounts for the EOF symbol as well, which is appended by the driver to mark the end of one transaction.

The RX-complete event data is defined in `rmt_rx_done_event_data_t`:

- `rmt_rx_done_event_data_t::received_symbols` points to the received RMT symbols. These symbols are saved in the `buffer` parameter of the `rmt_receive()` function. Users should not free this receive buffer before the callback returns.
- `rmt_rx_done_event_data_t::num_symbols` indicates the number of received RMT symbols. This value is not larger than the `buffer_size` parameter of `rmt_receive()` function. If the `buffer_size` is not sufficient to accommodate all the received RMT symbols, the driver only keeps the maximum number of symbols that the buffer can hold, and excess symbols are discarded or ignored.

Enable and Disable Channel  `rmt_enable()` must be called in advance before transmitting or receiving RMT symbols. For TX channels, enabling a channel enables a specific interrupt and prepares the hardware to dispatch transactions. For RX channels, enabling a channel enables an interrupt, but the receiver is not started during this time, as the characteristics of the incoming signal have yet to be specified. The receiver is started in `rmt_receive()`. `rmt_disable()` does the opposite by disabling the interrupt and clearing any pending interrupts. The transmitter and receiver are disabled as well.

```
ESP_ERROR_CHECK(rmt_enable(tx_chan));
ESP_ERROR_CHECK(rmt_enable(rx_chan));
```
Initiate TX Transaction  

RMT is a special communication peripheral, as it is unable to transmit raw byte streams like SPI and I2C. RMT can only send data in its own format `rmt_symbol_word_t`. However, the hardware does not help to convert the user data into RMT symbols, this can only be done in software by the so-called **RMT Encoder**. The encoder is responsible for encoding user data into RMT symbols and then writing to the RMT memory block or the DMA buffer. For how to create an RMT encoder, please refer to **RMT Encoder**.

Once you created an encoder, you can initiate a TX transaction by calling `rmt_transmit()`. This function takes several positional parameters like channel handle, encoder handle, and payload buffer. Besides, you also need to provide a transmission-specific configuration in `rmt_transmit_config_t`:

- `rmt_transmit_config_t::loop_count` sets the number of transmission loops. After the transmitter has finished one round of transmission, it can restart the same transmission again if this value is not set to zero. As the loop is controlled by hardware, the RMT channel can be used to generate many periodic sequences with minimal CPU intervention.
  - Setting `rmt_transmit_config_t::loop_count` to `-1` means an infinite loop transmission. In this case, the channel does not stop until `rmt_disable()` is called. The “trans-done” event is not generated as well.
  - Setting `rmt_transmit_config_t::loop_count` to a positive number means finite number of iterations. In this case, the “trans-done” event is when the specified number of iterations have completed.

  **Note:** The loop transmit feature is not supported on all ESP chips, please refer to [TRM] before you configure this option, or you might encounter `ESP_ERR_NOT_SUPPORTED` error.

- `rmt_transmit_config_t::eot_level` sets the output level when the transmitter finishes working or stops working by calling `rmt_disable()`.
- `rmt_transmit_config_t::queue_nonblocking` sets whether to wait for a free slot in the transaction queue when it is full. If this value is set to `true`, then the function will return with an error code `ESP_ERR_INVALID_STATE` when the queue is full. Otherwise, the function will block until a free slot is available in the queue.

  **Note:** There is a limitation in the transmission size if the `rmt_transmit_config_t::loop_count` is set to non-zero, i.e., to enable the loop feature. The encoded RMT symbols should not exceed the capacity of the RMT hardware memory block size, or you might see an error message like `encoding artifacts can't exceed hw memory block for loop transmission`. If you have to start a large transaction by loop, you can try either of the following methods.

  - Increase the `rmt_tx_channel_config_t::mem_block_symbols`. This approach does not work if the DMA backend is also enabled.
  - Customize an encoder and construct an infinite loop in the encoding function. See also **RMT Encoder**.

Internally, `rmt_transmit()` constructs a transaction descriptor and sends it to a job queue, which is dispatched in the ISR. So it is possible that the transaction is not started yet when `rmt_transmit()` returns. To ensure all pending transactions to complete, the user can use `rmt_tx_wait_all_done()`.

### Multiple Channels Simultaneous Transmission

In some real-time control applications (e.g., to make two robotic arms move simultaneously), you do not want any time drift between different channels. The RMT driver can help to manage this by creating a so-called **Sync Manager**. The sync manager is represented by `rmt_sync_manager_handle_t` in the driver. The procedure of RMT sync transmission is shown as follows:

**Install RMT Sync Manager**  

To create a sync manager, the user needs to tell which channels are going to be managed in the `rmt_sync_manager_config_t`:

- `rmt_sync_manager_config_t::tx_channel_array` points to the array of TX channels to be managed.
- `rmt_sync_manager_config_t::array_size` sets the number of channels to be managed.
Chapter 2. API Reference

![Fig. 17: RMT TX Sync](image)

`rmt_new_sync_manager()` can return a manager handle on success. This function could also fail due to various errors such as invalid arguments, etc. Especially, when the sync manager has been installed before, and there are no hardware resources to create another manager, this function reports `ESP_ERR_NOT_FOUND` error. In addition, if the sync manager is not supported by the hardware, it reports a `ESP_ERR_NOT_SUPPORTED` error. Please refer to [TRM] before using the sync manager feature.

**Start Transmission Simultaneously** For any managed TX channel, it does not start the machine until `rmt_transmit()` has been called on all channels in `rmt_sync_manager_config_t::tx_channel_array`. Before that, the channel is just put in a waiting state. TX channels will usually complete their transactions at different times due to differing transactions, thus resulting in a loss of sync. So before restarting a simultaneous transmission, the user needs to call `rmt_sync_reset()` to synchronize all channels again.

Calling `rmt_del_sync_manager()` can recycle the sync manager and enable the channels to initiate transactions independently afterward.

```c
rmt_channel_handle_t tx_channels[2] = {NULL}; // declare two channels
int tx_gpio_number[2] = {0, 2};
// install channels one by one
for (int i = 0; i < 2; i++) {
    rmt_tx_channel_config_t tx_chan_config = {
        .clk_src = RMT_CLK_SRC_DEFAULT, // select source clock
        .gpio_num = tx_gpio_number[i], // GPIO number
        .mem_block_symbols = 64, // memory block size, 64 * 4 = 256 Bytes
        .resolution_hz = 1 * 1000 * 1000, // 1 MHz resolution
        .trans_queue_depth = 1, // set the number of transactions that...
    };
    ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_chan_config, &tx_channels[i]));
}
// install sync manager
rmt_sync_manager_handle_t synchro = NULL;
rmt_sync_manager_config_t synchro_config = {
    .tx_channel_array = tx_channels,
    .array_size = sizeof(tx_channels) / sizeof(tx_channels[0]),
};
ESP_ERROR_CHECK(rmt_new_sync_manager(&synchro_config, &synchro));
ESP_ERROR_CHECK(rmt_transmit(tx_channels[0], led_strip_encoders[0], led_data, led_num * 3, transmit_config));
```

(continues on next page)
// tx_channels[0] does not start transmission until call of `rmt_transmit()` for tx_channels[1] returns
ESP_ERROR_CHECK(rmt_transmit(tx_channels[1], led_strip_encoders[1], led_data, led_num * 3, &transmit_config));

Initiate RX Transaction  As also discussed in the Enable and Disable Channel, calling `rmt_enable()` does not prepare an RX to receive RMT symbols. The user needs to specify the basic characteristics of the incoming signals in `rmt_receive_config_t`:

- `rmt_receive_config_t::signal_range_min_ns` specifies the minimal valid pulse duration in either high or low logic levels. A pulse width that is smaller than this value is treated as a glitch, and ignored by the hardware.
- `rmt_receive_config_t::signal_range_max_ns` specifies the maximum valid pulse duration in either high or low logic levels. A pulse width that is bigger than this value is treated as Stop Signal, and the receiver generates receive-complete event immediately.

The RMT receiver starts the RX machine after the user calls `rmt_receive()` with the provided configuration above. Note that, this configuration is transaction specific, which means, to start a new round of reception, the user needs to set the `rmt_receive_config_t` again. The receiver saves the incoming signals into its internal memory block or DMA buffer, in the format of `rmt_symbol_word_t`.

Due to the limited size of the memory block, the RMT receiver can only save short frames whose length is not longer than the memory block capacity. Long frames are truncated by the hardware, and the driver reports an error message: hw buffer too small, received symbols truncated.

The copy destination should be provided in the `buffer` parameter of `rmt_receive()` function. If this buffer overflows due to an insufficient buffer size, the receiver can continue to work, but overflowed symbols are dropped and the following error message is reported: user buffer too small, received symbols truncated. Please take care of the lifecycle of the `buffer` parameter, ensuring that the buffer is not recycled before the receiver is finished or stopped.

The receiver is stopped by the driver when it finishes working, i.e., receive a signal whose duration is bigger than `rmt_receive_config_t::signal_range_max_ns`. The user needs to call `rmt_receive()` again to restart the receiver, if necessary. The user can get the received data in the `rmt_rx_event_callbacks_t::on_recv_done` callback. See also Register Event Callbacks for more information.

```c
static bool example_rmt_rx_done_callback(rmt_channel_handle_t channel, const rmt_rx_done_event_data_t *edata, void *user_data)
{
    BaseType_t high_task_wakeup = pdFALSE;
    QueueHandle_t receive_queue = (QueueHandle_t)user_data;
    // send the received RMT symbols to the parser task
    xQueueSendFromISR(receive_queue, edata, &high_task_wakeup);
    // return whether any task is woken up
    return high_task_wakeup == pdTRUE;
}
```

```c
QueueHandle_t receive_queue = xQueueCreate(1, sizeof(rmt_rx_done_event_data_t));
rmt_rx_event_callbacks_t cbss = {
    .on_recv_done = example_rmt_rx_done_callback,
};
ESP_ERROR_CHECK(rmt_rx_register_event_callbacks(rx_channel, &cbss, receive_queue));
```

// the following timing requirement is based on NEC protocol
rmt_receive_config_t receive_config = {
    .signal_range_min_ns = 1250,  // the shortest duration for NEC signal is ~560 µs, 1250 ns < 560 µs, valid signal is not treated as noise
    .signal_range_max_ns = 12000000,  // the longest duration for NEC signal is ~9000 µs, 12000000 ns > 9000 µs, the receive does not stop early
}
RMT Encoder An RMT encoder is part of the RMT TX transaction, whose responsibility is to generate and write the correct RMT symbols into hardware memory or DMA buffer at a specific time. There are some special restrictions for an encoding function:

- During a single transaction, the encoding function may be called multiple times. This is necessary because the target RMT memory block cannot hold all the artifacts at once. To overcome this limitation, the driver utilizes a ping-pong approach, where the encoding session is divided into multiple parts. This means that the encoder needs to keep track of its state to continue encoding from where it left off in the previous part.
- The encoding function is running in the ISR context. To speed up the encoding session, it is highly recommended to put the encoding function into IRAM. This can also avoid the cache miss during encoding.

To help get started with the RMT driver faster, some commonly used encoders are provided out-of-the-box. They can either work alone or be chained together into a new encoder. See also Composite Pattern for the principle behind it. The driver has defined the encoder interface in `rmt_encoder_t`, it contains the following functions:

- `rmt_encoder_t::encode` is the fundamental function of an encoder. This is where the encoding session happens.
  - The function might be called multiple times within a single transaction. The encode function should return the state of the current encoding session.
  - The supported states are listed in the `rmt_encode_state_t`. If the result contains `RMT_ENCODING_COMPLETE`, it means the current encoder has finished work.
  - If the result contains `RMT_ENCODING_MEM_FULL`, the program needs to yield from the current session, as there is no space to save more encoding artifacts.
- `rmt_encoder_t::reset` should reset the encoder state back to the initial state (the RMT encoder is stateful).
  - If the RMT transmitter is manually stopped without resetting its corresponding encoder, subsequent encoding session can be erroneous.
  - This function is also called implicitly in `rmt_disable()`.
- `rmt_encoder_t::del` should free the resources allocated by the encoder.

Copy Encoder A copy encoder is created by calling `rmt_new_copy_encoder()`. A copy encoder’s main functionality is to copy the RMT symbols from user space into the driver layer. It is usually used to encode `const` data, i.e., data does not change at runtime after initialization such as the leading code in the IR protocol.

A configuration structure `rmt_copy_encoder_config_t` should be provided in advance before calling `rmt_new_copy_encoder()`. Currently, this configuration is reserved for future expansion, and has no specific use or setting items for now.

Bytes Encoder A bytes encoder is created by calling `rmt_new_bytes_encoder()`. The bytes encoder’s main functionality is to convert the user space byte stream into RMT symbols dynamically. It is usually used to encode dynamic data, e.g., the address and command fields in the IR protocol.

A configuration structure `rmt_bytes_encoder_config_t` should be provided in advance before calling `rmt_new_bytes_encoder()`: 

```c
rmt_symbol_word_t raw_symbols[64]; // 64 symbols should be sufficient for a_
standard NEC frame
// ready to receive
ESP_ERROR_CHECK(rmt_receive(rx_channel, raw_symbols, sizeof(raw_symbols), &receive_ --config));
// wait for the RX-done signal
rmt_rx_done_event_data_t rx_data;
xQueueReceive(receive_queue, &rx_data, portMAX_DELAY);
// parse the received symbols
example_parse_nec_frame(rx_data.received_symbols, rx_data.num_symbols);
```
• `rmt_bytes_encoder_config_t::bit0` and `rmt_bytes_encoder_config_t::bit1` are necessary to specify the encoder how to represent bit zero and bit one in the format of `rmt_symbol_word_t`.

• `rmt_bytes_encoder_config_t::msb_first` sets the bit endianness of each byte. If it is set to true, the encoder encodes the Most Significant Bit first. Otherwise, it encodes the Least Significant Bit first.

Besides the primitive encoders provided by the driver, the user can implement his own encoder by chaining the existing encoders together. A common encoder chain is shown as follows:

![Diagram of RMT Encoder Chain](image)

**Fig. 18: RMT Encoder Chain**

**Customize RMT Encoder for NEC Protocol** This section demonstrates how to write an NEC encoder. The NEC IR protocol uses pulse distance encoding of the message bits. Each pulse burst is 562.5 µs in length, logical bits are transmitted as follows. It is worth mentioning that the least significant bit of each byte is sent first.

- Logical 0: a 562.5 µs pulse burst followed by a 562.5 µs space, with a total transmit time of 1.125 ms
- Logical 1: a 562.5 µs pulse burst followed by a 1.6875 ms space, with a total transmit time of 2.25 ms

When a key is pressed on the remote controller, the transmitted message includes the following elements in the specified order:

![IR NEC Frame](image)

**Fig. 19: IR NEC Frame**

• 9 ms leading pulse burst, also called the “AGC pulse”
• 4.5 ms space
• 8-bit address for the receiving device
• 8-bit logical inverse of the address
• 8-bit command
• 8-bit logical inverse of the command
• a final 562.5 µs pulse burst to signify the end of message transmission

Then you can construct the NEC `rmt_encoder_t::encode` function in the same order, for example:
// IR NEC scan code representation
typedef struct {
    uint16_t address;
    uint16_t command;
} ir_nec_scan_code_t;

// construct an encoder by combining primitive encoders
typedef struct {
    rmt_encoder_t base;     // the base "class" declares the standard
    rmt_encoder_t *copy_encoder; // use the copy_encoder to encode the leading
    rmt_encoder_t *bytes_encoder; // use the bytes_encoder to encode the address
    rmt_symbol_word_t nec_leading_symbol; // NEC leading code with RMT
    rmt_symbol_word_t nec_ending_symbol; // NEC ending code with RMT
    int state;     // record the current encoding state, i.e., we are in which
                   // encoding phase
} rmt_ir_nec_encoder_t;

static size_t rmt_encode_ir_nec(rmt_encoder_t *encoder, rmt_channel_handle_t channel, const void *primary_data, size_t data_size, rmt_encode_state_t *ret_state) {
    rmt_ir_nec_encoder_t *nec_encoder = __containerof(encoder, rmt_ir_nec_encoder_t, base);
    rmt_encode_state_t session_state = RMT_ENCODING_RESET;
    rmt_encode_state_t state = RMT_ENCODING_RESET;
    size_t encoded_symbols = 0;
    ir_nec_scan_code_t *scan_code = (ir_nec_scan_code_t *)primary_data;
    rmt_encoder_handle_t copy_encoder = nec_encoder->copy_encoder;
    rmt_encoder_handle_t bytes_encoder = nec_encoder->bytes_encoder;
    switch (nec_encoder->state) {
    case 0: // send leading code
        encoded_symbols += copy_encoder->encode(copy_encoder, channel, &nec_encoder->nec_leading_symbol, sizeof(rmt_symbol_word_t), &session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            nec_encoder->state = 1; // we can only switch to the next state when
                                    // the current encoder finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there is no free space to put other encoding
        }
        break;
    case 1: // send address
        encoded_symbols += bytes_encoder->encode(bytes_encoder, channel, &scan_code->address, sizeof(uint16_t), &session_state);
        if (session_state & RMT_ENCODING_COMPLETE) {
            nec_encoder->state = 2; // we can only switch to the next state when
                                    // the current encoder finished
        }
        if (session_state & RMT_ENCODING_MEM_FULL) {
            state |= RMT_ENCODING_MEM_FULL;
            goto out; // yield if there is no free space to put other encoding
        }
    }
    // fall-through
    // construct an encoder by combining primitive encoders
    // the base "class" declares the standard
    // use the copy_encoder to encode the leading
    // use the bytes_encoder to encode the address
    // NEC leading code with RMT
    // NEC ending code with RMT
    // record the current encoding state, i.e., we are in which
    // encoding phase
    }
// fall-through
  case 2: // send command
    encoded_symbols += bytes_encoder->encode(bytes_encoder, channel, &scan_
    ->command, sizeof(uint16_t), &session_state);
    if (session_state & RMT_ENCODING_COMPLETE) {
      nec_encoder->state = 3; // we can only switch to the next state when
      // the current encoder finished
    }
    if (session_state & RMT_ENCODING_MEM_FULL) {
      state |= RMT_ENCODING_MEM_FULL;
      goto out; // yield if there is no free space to put other encoding...
    } goto artifacts;
  }
  // fall-through
  case 3: // send ending code
    encoded_symbols += copy_encoder->encode(copy_encoder, channel, &nec_
    ->encoder->nec_ending_symbol, sizeof(rmt_symbol_word_t), &
    ->session_state);
    if (session_state & RMT_ENCODING_COMPLETE) {
      nec_encoder->state = RMT_ENCODING_RESET; // back to the initial...
      encoding session
      state |= RMT_ENCODING_COMPLETE; // telling the caller the NEC encoding...
      has finished
    }
    if (session_state & RMT_ENCODING_MEM_FULL) {
      state |= RMT_ENCODING_MEM_FULL;
      goto out; // yield if there is no free space to put other encoding...
    } goto artifacts;
  }
out:
  *ret_state = state;
  return encoded_symbols;
}

A full sample code can be found in peripherals/rmt/ir_nec_transceiver. In the above snippet, we use a
switch-case and several goto statements to implement a Finite-state machine. With this pattern, users can
construct much more complex IR protocols.

Power Management  When power management is enabled, i.e., CONFIG_PM_ENABLE is on, the system adjusts
the APB frequency before going into Light-sleep, thus potentially changing the resolution of the RMT internal counter.
However, the driver can prevent the system from changing APB frequency by acquiring a power manage-
ment lock of type ESP_PM_APB_FREQ_MAX. Whenever the user creates an RMT channel that has selected
RMT_CLK_SRC_APB as the clock source, the driver guarantees that the power management lock is acquired af-
ter the channel enabled by rmt_enable(). Likewise, the driver releases the lock after rmt_disable() is
called for the same channel. This also reveals that the rmt_enable() and rmt_disable() should appear in
pairs.
If the channel clock source is selected to others like RMT_CLK_SRC_XTAL, then the driver does not install a power
management lock for it, which is more suitable for a low-power application as long as the source clock can still provide
sufficient resolution.

IRAM Safe  By default, the RMT interrupt is deferred when the Cache is disabled for reasons like writing or erasing
the main Flash. Thus the transaction-done interrupt does not get handled in time, which is not acceptable in a real-
time application. What is worse, when the RMT transaction relies on ping-pong interrupt to successively encode or
copy RMT symbols, a delayed interrupt can lead to an unpredictable result.
There is a Kconfig option CONFIG_RMT_ISR_IRAM_SAFE that has the following features:
1. Enable the interrupt being serviced even when the cache is disabled
2. Place all functions used by the ISR into IRAM
3. Place the driver object into DRAM in case it is mapped to PSRAM by accident

This Kconfig option allows the interrupt handler to run while the cache is disabled but comes at the cost of increased IRAM consumption.

Another Kconfig option `CONFIG_RMT_RECV_FUNC_IN_IRAM` can place `rmt_receive()` into the IRAM as well. So that the receive function can be used even when the flash cache is disabled.

**Thread Safety**

The factory function `rmt_new_tx_channel()`, `rmt_new_rx_channel()` and `rmt_new_sync_manager()` are guaranteed to be thread-safe by the driver, which means, user can call them from different RTOS tasks without protection by extra locks. Other functions that take the `rmt_channel_handle_t` and `rmt_sync_manager_handle_t` as the first positional parameter, are not thread-safe. Which means the user should avoid calling them from multiple tasks.

The following functions are allowed to use under ISR context as well.

- `rmt_receive()`

**Kconfig Options**

- `CONFIG_RMT_ISR_IRAM_SAFE` controls whether the default ISR handler can work when cache is disabled, see also `IRAM Safe` for more information.
- `CONFIG_RMT_ENABLE_DEBUG_LOG` is used to enable the debug log at the cost of increased firmware binary size.
- `CONFIG_RMT_RECV_FUNC_IN_IRAM` controls where to place the RMT receive function (IRAM or Flash), see `IRAM Safe` for more information.

**Application Examples**

- RMT-based RGB LED strip customized encoder: `peripherals/rmt/led_strip`
- RMT IR NEC protocol encoding and decoding: `peripherals/rmt/ir_nec_transceiver`
- RMT transactions in queue: `peripherals/rmt/musical_buzzer`
- RMT-based stepper motor with S-curve algorithm: `peripherals/rmt/stepper_motor`
- RMT infinite loop for driving DShot ESC: `peripherals/rmt/dshot_esc`
- RMT simulate 1-wire protocol (take DS18B20 as example): `peripherals/rmt/onewire`

**FAQ**

- Why the RMT encoder results in more data than expected?

The RMT encoding takes place in the ISR context. If your RMT encoding session takes a long time (e.g., by logging debug information) or the encoding session is deferred somehow because of interrupt latency, then it is possible the transmitting becomes faster than the encoding. As a result, the encoder can not prepare the next data in time, leading to the transmitter sending the previous data again. There is no way to ask the transmitter to stop and wait. You can mitigate the issue by combining the following ways:

- Increase the `rmt_tx_channel_config_t::mem_block_symbols`, in steps of 64.
- Place the encoding function in the IRAM.
- Enables the `rmt_tx_channel_config_t::with_dma` if it is available for your chip.

---

2 The callback function, e.g., `rmt_tx_event_callbacks_t::on_trans_done`, and the functions invoked by itself should also reside in IRAM, users need to take care of this by themselves.
API Reference

Header File

- components/driver/rmt/include/driver/rmt_tx.h
- This header file can be included with:

```
#include "driver/rmt_tx.h"
```
- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```
REQUIRES driver
```

or

```
PRIV_REQUIRES driver
```

Functions

**esp_err_t rmt_new_tx_channel** (const rmt_tx_channel_config_t *config, rmt_channel_handle_t *ret_chan)

Create a RMT TX channel.

**Parameters**

- config ⚶ [in] TX channel configurations
- ret_chan ⚶ [out] Returned generic RMT channel handle

**Returns**

- ESP_OK: Create RMT TX channel successfully
- ESP_ERR_INVALID_ARG: Create RMT TX channel failed because of invalid argument
- ESP_ERR_NO_MEM: Create RMT TX channel failed because of out of memory
- ESP_ERR_NOT_FOUND: Create RMT TX channel failed because all RMT channels are used up and no more free one
- ESP_ERR_NOT_SUPPORTED: Create RMT TX channel failed because some feature is not supported by hardware, e.g. DMA feature is not supported by hardware
- ESP_FAIL: Create RMT TX channel failed because of other error

**esp_err_t rmt_transmit** (rmt_channel_handle_t tx_channel, rmt_encoder_handle_t encoder, const void *payload, size_t payload_bytes, const rmt_transmit_config_t *config)

Transmit data by RMT TX channel.

**Note:** This function constructs a transaction descriptor then pushes to a queue. The transaction will not start immediately if there’s another one under processing. Based on the setting of rmt_transmit_config_t::queue_nonblocking, if there’re too many transactions pending in the queue, this function can block until it has free slot, otherwise just return quickly.

**Note:** The data to be transmitted will be encoded into RMT symbols by the specific encoder.

**Parameters**

- tx_channel ⚶ [in] RMT TX channel that created by rmt_new_tx_channel()
- encoder ⚶ [in] RMT encoder that created by various factory APIs like rmt_new_bytes_encoder()
- payload ⚶ [in] The raw data to be encoded into RMT symbols
- payload_bytes ⚶ [in] Size of the payload in bytes
- config ⚶ [in] Transmission specific configuration

**Returns**

- ESP_OK: Transmit data successfully
- ESP_ERR_INVALID_ARG: Transmit data failed because of invalid argument
- ESP_FAIL: Transmit data failed because channel is not enabled
• ESP_ERR_NOT_SUPPORTED: Transmit data failed because some feature is not supported by hardware, e.g. unsupported loop count
• ESP_FAIL: Transmit data failed because of other error

`esp_err_t rmt_tx_wait_all_done (rmt_channel_handle_t tx_channel, int timeout_ms)`
Wait for all pending TX transactions done.

**Note:** This function will block forever if the pending transaction can’t be finished within a limited time (e.g. an infinite loop transaction). See also `rmt_disable()` for how to terminate a working channel.

**Parameters**
- `tx_channel` - [in] RMT TX channel that created by `rmt_new_tx_channel()`

**Returns**
- ESP_OK: Flush transactions successfully
- ESP_ERR_INVALID_ARG: Flush transactions failed because of invalid argument
- ESP_ERR_TIMEOUT: Flush transactions failed because of timeout
- ESP_FAIL: Flush transactions failed because of other error

`esp_err_t rmt_tx_register_event_callbacks (rmt_channel_handle_t tx_channel, const rmt_tx_event_callbacks_t *cbs, void *user_data)`
Set event callbacks for RMT TX channel.

**Note:** User can deregister a previously registered callback by calling this function and setting the callback member in the `cbs` structure to NULL.

**Note:** When `CONFIG_RMT_ISR_IRAM_SAFE` is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

**Parameters**
- `tx_channel` - [in] RMT generic channel that created by `rmt_new_tx_channel()`
- `cbs` - [in] Group of callback functions
- `user_data` - [in] User data, which will be passed to callback functions directly

**Returns**
- ESP_OK: Set event callbacks successfully
- ESP_ERR_INVALID_ARG: Set event callbacks failed because of invalid argument
- ESP_FAIL: Set event callbacks failed because of other error

`esp_err_t rmt_new_sync_manager (const rmt_sync_manager_config_t *config, rmt_sync_manager_handle_t *ret_synchro)`
Create a synchronization manager for multiple TX channels, so that the managed channel can start transmitting at the same time.

**Note:** All the channels to be managed should be enabled by `rmt_enable()` before put them into sync manager.

**Parameters**
- `config` - [in] Synchronization manager configuration
- `ret_synchro` - [out] Returned synchronization manager handle

**Returns**
Chapter 2. API Reference

- ESP_OK: Create sync manager successfully
- ESP_ERR_INVALID_ARG: Create sync manager failed because of invalid argument
- ESP_ERR_NOT_SUPPORTED: Create sync manager failed because it is not supported by hardware
- ESP_ERR_INVALID_STATE: Create sync manager failed because not all channels are enabled
- ESP_ERR_NO_MEM: Create sync manager failed because out of memory
- ESP_ERR_NOT_FOUND: Create sync manager failed because all sync controllers are used up and no more free one
- ESP_FAIL: Create sync manager failed because of other error

```c
esp_err_t rmt_del_sync_manager(rmt_sync_manager_handle_t synchro)
```

Delete synchronization manager.

**Parameters**
- `synchro` — [in] Synchronization manager handle returned from `rmt_new_sync_manager()`

**Returns**
- ESP_OK: Delete the synchronization manager successfully
- ESP_ERR_INVALID_ARG: Delete the synchronization manager failed because of invalid argument
- ESP_FAIL: Delete the synchronization manager failed because of other error

```c
esp_err_t rmt_sync_reset(rmt_sync_manager_handle_t synchro)
```

Reset synchronization manager.

**Parameters**
- `synchro` — [in] Synchronization manager handle returned from `rmt_new_sync_manager()`

**Returns**
- ESP_OK: Reset the synchronization manager successfully
- ESP_ERR_INVALID_ARG: Reset the synchronization manager failed because of invalid argument
- ESP_FAIL: Reset the synchronization manager failed because of other error

**Structures**

```c
struct rmt_tx_event_callbacks_t
```

Group of RMT TX callbacks.

**Note:** The callbacks are all running under ISR environment

**Note:** When `CONFIG_RMT_ISR_IRAM_SAFE` is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

**Public Members**

```c
rmt_tx_done_callback_t on_trans_done
```

Event callback, invoked when transmission is finished

```c
struct rmt_tx_channel_config_t
```

RMT TX channel specific configuration.

**Public Members**
### Chapter 2. API Reference

**gpio_num_t**

- **gpio_num**
  - GPIO number used by RMT TX channel. Set to -1 if unused

**rmt_clock_source_t**

- **clk_src**
  - Clock source of RMT TX channel, channels in the same group must use the same clock source

**uint32_t**

- **resolution_hz**
  - Channel clock resolution, in Hz

**size_t**

- **mem_block_symbols**
  - Size of memory block, in number of `rmt_symbol_word_t`, must be an even. In the DMA mode, this field controls the DMA buffer size, it can be set to a large value; In the normal mode, this field controls the number of RMT memory block that will be used by the channel.

- **trans_queue_depth**
  - Depth of internal transfer queue, increase this value can support more transfers pending in the background

**int**

- **intr_priority**
  - RMT interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

- **invert_out**
  - Whether to invert the RMT channel signal before output to GPIO pad

- **with_dma**
  - If set, the driver will allocate an RMT channel with DMA capability

- **io_loop_back**
  - The signal output from the GPIO will be fed to the input path as well

- **io_od_mode**
  - Configure the GPIO as open-drain mode

**struct**

- **rmt_tx_channel_config_t**: [anonymous] **flags**
  - TX channel config flags

- **rmt_transmit_config_t**
  - RMT transmit specific configuration.

#### Public Members

**int**

- **loop_count**
  - Specify the times of transmission in a loop, -1 means transmitting in an infinite loop

- **eot_level**
  - Set the output level for the “End Of Transmission”
Chapter 2. API Reference

uint32_t queue_nonblocking

If set, when the transaction queue is full, driver will not block the thread but return directly

struct rmt_transmit_config_t::[anonymous] flags

Transmit specific config flags

struct rmt_sync_manager_config_t

Synchronous manager configuration.

Public Members

const rmt_channel_handle_t *tx_channel_array

Array of TX channels that are about to be managed by a synchronous controller

size_t array_size

Size of the tx_channel_array

Header File

• components/driver/rmt/include/driver/rmt_rx.h
• This header file can be included with:

```
#include "driver/rmt_rx.h"
```

• This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```
REQUIRES driver
```

or

```
PRIV_REQUIRES driver
```

Functions

`esp_err_t rmt_new_rx_channel` (const rmt_rx_channel_config_t *config, rmt_channel_handle_t *ret_chan)

Create a RMT RX channel.

Parameters

• config [in] RX channel configurations
• ret_chan [out] Returned generic RMT channel handle

Returns

• ESP_OK: Create RMT RX channel successfully
• ESP_ERR_INVALID_ARG: Create RMT RX channel failed because of invalid argument
• ESP_ERR_NO_MEM: Create RMT RX channel failed because out of memory
• ESP_ERR_NOT_FOUND: Create RMT RX channel failed because all RMT channels are used up and no more free one
• ESP_ERR_NOT_SUPPORTED: Create RMT RX channel failed because some feature is not supported by hardware, e.g. DMA feature is not supported by hardware
• ESP_FAIL: Create RMT RX channel failed because of other error

`esp_err_t rmt_receive` (rmt_channel_handle_t rx_channel, void *buffer, size_t buffer_size, const rmt_receive_config_t *config)

Initiate a receive job for RMT RX channel.
**Note:** This function is non-blocking, it initiates a new receive job and then returns. User should check the received data from the `on_recv_done` callback that registered by `rmt_rx_register_event_callbacks()`.

**Note:** This function can also be called in ISR context.

**Note:** If you want this function to work even when the flash cache is disabled, please enable the `CONFIG_RMT_RECV_FUNC_IN_IRAM` option.

### Parameters
- `rx_channel` - [in] RMT RX channel that created by `rmt_new_rx_channel()`
- `buffer` - [in] The buffer to store the received RMT symbols
- `buffer_size` - [in] size of the `buffer`, in bytes
- `config` - [in] Receive specific configurations

### Returns
- `ESP_OK`: Initiate receive job successfully
- `ESP_ERR_INVALID_ARG`: Initiate receive job failed because of invalid argument
- `ESP_ERR_INVALID_STATE`: Initiate receive job failed because channel is not enabled
- `ESP_FAIL`: Initiate receive job failed because of other error

```c
esp_err_t rmt_rx_register_event_callbacks(rmt_channel_handle_t rx_channel, const rmt_rx_event_callbacks_t *cbs, void *user_data)
```

Set callbacks for RMT RX channel.

**Note:** User can deregister a previously registered callback by calling this function and setting the `callback` member in the `cbs` structure to NULL.

**Note:** When `CONFIG_RMT_ISR_IRAM_SAFE` is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well. The `user_data` should also reside in SRAM.

### Parameters
- `rx_channel` - [in] RMT generic channel that created by `rmt_new_rx_channel()`
- `cbs` - [in] Group of callback functions
- `user_data` - [in] User data, which will be passed to callback functions directly

### Returns
- `ESP_OK`: Set event callbacks successfully
- `ESP_ERR_INVALID_ARG`: Set event callbacks failed because of invalid argument
- `ESP_FAIL`: Set event callbacks failed because of other error

### Structures
```c
struct rmt_rx_event_callbacks_t
```

Group of RMT RX callbacks.

**Note:** The callbacks are all running under ISR environment.
Note: When CONFIG_RMT_ISR_IRAM_SAFE is enabled, the callback itself and functions called by it should be placed in IRAM. The variables used in the function should be in the SRAM as well.

**Public Members**

*rmt_rx_done_callback_t on_recv_done*

Event callback, invoked when one RMT channel receiving transaction completes

**struct rmt_rx_channel_config_t**

RMT RX channel specific configuration.

**Public Members**

*gpio_num_t gpio_num*

GPIO number used by RMT RX channel. Set to -1 if unused

*rmt_clock_source_t clk_src*

Clock source of RMT RX channel, channels in the same group must use the same clock source

*uint32_t resolution_hz*

Channel clock resolution, in Hz

*size_t mem_block_symbols*

Size of memory block, in number of *rmt_symbol_word_t*, must be an even. In the DMA mode, this field controls the DMA buffer size, it can be set to a large value (e.g. 1024); In the normal mode, this field controls the number of RMT memory block that will be used by the channel.

*uint32_t invert_in*

Whether to invert the incoming RMT channel signal

*uint32_t with_dma*

If set, the driver will allocate an RMT channel with DMA capability

*uint32_t io_loop_back*

For debug/test, the signal output from the GPIO will be fed to the input path as well

**struct rmt_rx_channel_config_t::[anonymous] flags**

RX channel config flags

*int intr_priority*

RMT interrupt priority, if set to 0, the driver will try to allocate an interrupt with a relative low priority (1,2,3)

**struct rmt_receive_config_t**

RMT receive specific configuration.
Public Members

\textit{uint32_t} \texttt{signal\_range\_min\_ns}

A pulse whose width is smaller than this threshold will be treated as glitch and ignored

\textit{uint32_t} \texttt{signal\_range\_max\_ns}

RMT will stop receiving if one symbol level has kept more than \texttt{signal\_range\_max\_ns}

Header File

- \textit{components/driver/rmt/include/driver/rmt_common.h}
- This header file can be included with:

\begin{verbatim}
#include "driver/rmt_common.h"
\end{verbatim}

- This header file is a part of the API provided by the \textit{driver} component. To declare that your component depends on \textit{driver}, add the following to your CMakeLists.txt:

\begin{verbatim}
REQUIRES driver
\end{verbatim}

or

\begin{verbatim}
PRIV_REQUIRES driver
\end{verbatim}

Functions

\textit{esp_err_t rmt_del_channel} (\textit{rmt\_channel\_handle\_t channel})

Delete an RMT channel.

Parameters \texttt{channel} \texttt{[in]} RMT generic channel that created by \texttt{rmt\_new\_tx\_channel()} or \texttt{rmt\_new\_rx\_channel()}

Returns

- ESP\_OK: Delete RMT channel successfully
- ESP\_ERR\_INVALID\_ARG: Delete RMT channel failed because of invalid argument
- ESP\_ERR\_INVALID\_STATE: Delete RMT channel failed because it is still in working
- ESP\_FAIL: Delete RMT channel failed because of other error

\textit{esp_err_t rmt\_apply\_carrier} (\textit{rmt\_channel\_handle\_t channel}, \textit{const rmt\_carrier\_config\_t *config})

Apply modulation feature for TX channel or demodulation feature for RX channel.

Parameters

- \texttt{channel} \texttt{[in]} RMT generic channel that created by \texttt{rmt\_new\_tx\_channel()} or \texttt{rmt\_new\_rx\_channel()}
- \texttt{config} \texttt{[in]} Carrier configuration. Specially, a NULL config means to disable the carrier modulation or demodulation feature

Returns

- ESP\_OK: Apply carrier configuration successfully
- ESP\_ERR\_INVALID\_ARG: Apply carrier configuration failed because of invalid argument
- ESP\_FAIL: Apply carrier configuration failed because of other error

\textit{esp_err_t rmt\_enable} (\textit{rmt\_channel\_handle\_t channel})

Enable the RMT channel.

\textbf{Note:} This function will acquire a PM lock that might be installed during channel allocation

Parameters \texttt{channel} \texttt{[in]} RMT generic channel that created by \texttt{rmt\_new\_tx\_channel()} or \texttt{rmt\_new\_rx\_channel()}
Chapter 2. API Reference

Returns

- ESP_OK: Enable RMT channel successfully
- ESP_ERR_INVALID_ARG: Enable RMT channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable RMT channel failed because it’s enabled already
- ESP_FAIL: Enable RMT channel failed because of other error

```c
esp_err_t rmt_disable(rmt_channel_handle_t channel)
```

Disable the RMT channel.

Note: This function will release a PM lock that might be installed during channel allocation

Parameters

channel - [in] RMT generic channel that created by rmt_new_tx_channel() or rmt_new_rx_channel()

Returns

- ESP_OK: Disable RMT channel successfully
- ESP_ERR_INVALID_ARG: Disable RMT channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable RMT channel failed because it’s not enabled yet
- ESP_FAIL: Disable RMT channel failed because of other error

Structures

struct rmt_carrier_config_t

RMT carrier wave configuration (for either modulation or demodulation)

Public Members

`uint32_t frequency_hz`

Carrier wave frequency, in Hz, 0 means disabling the carrier

`float duty_cycle`

Carrier wave duty cycle (0-100%)

`uint32_t polarity_active_low`

Specify the polarity of carrier, by default it’s modulated to base signal’s high level

`uint32_t always_on`

If set, the carrier can always exist even there’s not transfer undergoing

.struct rmt_carrier_config_t::[anonymous] flags

Carrier config flags

Header File

- components/driver/rmt/include/driver/rmt_encoder.h
- This header file can be included with:

```c
#include "driver/rmt_encoder.h"
```
- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
Functions

.esp_err_t rmt_new_bytes_encoder (const rmt_bytes_encoder_config_t *config, rmt_encoder_handle_t *ret_encoder)

Create RMT bytes encoder, which can encode byte stream into RMT symbols.

Parameters

• config - [in] Bytes encoder configuration
• ret_encoder - [out] Returned encoder handle

Returns

• ESP_OK: Create RMT bytes encoder successfully
• ESP_ERR_INVALID_ARG: Create RMT bytes encoder failed because of invalid argument
• ESP_ERR_NO_MEM: Create RMT bytes encoder failed because out of memory
• ESP_FAIL: Create RMT bytes encoder failed because of other error

.esp_err_t rmt_new_copy_encoder (const rmt_copy_encoder_config_t *config, rmt_encoder_handle_t *ret_encoder)

Create RMT copy encoder, which copies the given RMT symbols into RMT memory.

Parameters

• config - [in] Copy encoder configuration
• ret_encoder - [out] Returned encoder handle

Returns

• ESP_OK: Create RMT copy encoder successfully
• ESP_ERR_INVALID_ARG: Create RMT copy encoder failed because of invalid argument
• ESP_ERR_NO_MEM: Create RMT copy encoder failed because out of memory
• ESP_FAIL: Create RMT copy encoder failed because of other error

.esp_err_t rmt_del_encoder (rmt_encoder_handle_t encoder)

Delete RMT encoder.

Parameters encoder - [in] RMT encoder handle, created by e.g.

rmt_new_bytes_encoder()

Returns

• ESP_OK: Delete RMT encoder successfully
• ESP_ERR_INVALID_ARG: Delete RMT encoder failed because of invalid argument
• ESP_FAIL: Delete RMT encoder failed because of other error

.esp_err_t rmt_encoder_reset (rmt_encoder_handle_t encoder)

Reset RMT encoder.

Parameters encoder - [in] RMT encoder handle, created by e.g.

rmt_new_bytes_encoder()

Returns

• ESP_OK: Reset RMT encoder successfully
• ESP_ERR_INVALID_ARG: Reset RMT encoder failed because of invalid argument
• ESP_FAIL: Reset RMT encoder failed because of other error

Structures

struct rmt_encoder_t

Interface of RMT encoder.
Public Members

size_t (*encode)(rmt_encoder_t *encoder, rmt_channel_handle_t tx_channel, const void *primary_data, size_t data_size, rmt_encode_state_t *ret_state)

Encode the user data into RMT symbols and write into RMT memory.

**Note:** The encoding function will also be called from an ISR context, thus the function must not call any blocking API.

**Note:** It’s recommended to put this function implementation in the IRAM, to achieve a high performance and less interrupt latency.

### Param encoder [in] Encoder handle

### Param tx_channel [in] RMT TX channel handle, returned from `rmt_new_tx_channel()`

### Param primary_data [in] App data to be encoded into RMT symbols

### Param data_size [in] Size of primary_data, in bytes

### Param ret_state [out] Returned current encoder’s state

### Return Number of RMT symbols that the primary data has been encoded into

esp_err_t (*reset)(rmt_encoder_t *encoder)

Reset encoding state.

### Param encoder [in] Encoder handle

### Return

- ESP_OK: reset encoder successfully
- ESP_FAIL: reset encoder failed

espt_t (*del)(rmt_encoder_t *encoder)

Delete encoder object.

### Param encoder [in] Encoder handle

### Return

- ESP_OK: delete encoder successfully
- ESP_FAIL: delete encoder failed

struct rmt_bytes_encoder_config_t

Bytes encoder configuration.

Public Members

rmt_symbol_word_t bit0

How to represent BIT0 in RMT symbol

rmt_symbol_word_t bit1

How to represent BIT1 in RMT symbol

uint32_t msb_first

Whether to encode MSB bit first
struct `rmt_bytes_encoder_config_t`::[anonymous] **flags**
Encoder config flag

struct **rmt_copy_encoder_config_t**
Copy encoder configuration.

**Enumerations**

enum **rmt_encode_state_t**
RMT encoding state.

*Values:*

enumerator **RMT.Encoding_RESET**
The encoding session is in reset state

enumerator **RMT.Encoding_COMPLETE**
The encoding session is finished, the caller can continue with subsequent encoding

enumerator **RMT.Encoding_MEM_FULL**
The encoding artifact memory is full, the caller should return from current encoding session

**Header File**

- components/driver/rmt/include/driver/rmt_types.h
- This header file can be included with:

  ```
  #include "driver/rmt_types.h"
  ```

- This header file is a part of the API provided by the **driver** component. To declare that your component depends on **driver**, add the following to your CMakeLists.txt:

  ```
  REQUIRES driver
  ```
  or

  ```
  PRIV_REQUIRES driver
  ```

**Structures**

struct **rmt_tx_done_event_data_t**
Type of RMT TX done event data.

**Public Members**

size_t **num_symbols**
The number of transmitted RMT symbols, including one EOF symbol, which is appended by the driver to mark the end of a transmission. For a loop transmission, this value only counts for one round.

struct **rmt_rx_done_event_data_t**
Type of RMT RX done event data.
Public Members

\[ \textit{rmt\_symbol\_word\_t \ast received\_symbols} \]

Point to the received RMT symbols

\[ \textit{size\_t num\_symbols} \]

The number of received RMT symbols

Type Definitions

typedef struct rmt\_channel\_t *rmt\_channel\_handle\_t

Type of RMT channel handle.

typedef struct rmt\_sync\_manager\_t *rmt\_sync\_manager\_handle\_t

Type of RMT synchronization manager handle.

typedef struct rmt\_encoder\_t *rmt\_encoder\_handle\_t

Type of RMT encoder handle.

typedef bool (*rmt\_tx\_done\_callback\_t)(rmt\_channel\_handle\_t tx\_chan, const rmt\_tx\_done\_event\_data\_t *edata, void *user\_ctx)

Prototype of RMT event callback.

- \textbf{Param tx\_chan [in]} RMT channel handle, created from rmt\_new\_tx\_channel()
- \textbf{Param edata [in]} Point to RMT event data. The lifecycle of this pointer memory is inside this function, user should copy it into static memory if used outside this function.
- \textbf{Param user\_ctx [in]} User registered context, passed from rmt\_tx\_register\_event\_callbacks()

\textbf{Return} Whether a high priority task has been waken up by this callback function

typedef bool (*rmt\_rx\_done\_callback\_t)(rmt\_channel\_handle\_t rx\_chan, const rmt\_rx\_done\_event\_data\_t *edata, void *user\_ctx)

Prototype of RMT event callback.

- \textbf{Param rx\_chan [in]} RMT channel handle, created from rmt\_new\_rx\_channel()
- \textbf{Param edata [in]} Point to RMT event data. The lifecycle of this pointer memory is inside this function, user should copy it into static memory if used outside this function.
- \textbf{Param user\_ctx [in]} User registered context, passed from rmt\_rx\_register\_event\_callbacks()

\textbf{Return} Whether a high priority task has been waken up by this function

Header File

- components/hal/include/hal/rmt\_types.h
- This header file can be included with:

```c
#include "hal/rmt\_types.h"
```

Unions

union rmt\_symbol\_word\_t

#include <rmt\_types.h> The layout of RMT symbol stored in memory, which is decided by the hardware design.
Chapter 2. API Reference

Public Members

```c
uint16_t duration0
   Duration of level0

uint16_t level0
   Level of the first part

uint16_t duration1
   Duration of level1

uint16_t level1
   Level of the second part

struct rmt_symbol_word_t::[anonymous] [anonymous]

uint32_t val
   Equivalent unsigned value for the RMT symbol
```

Type Definitions

typedef soc_periph_rmt_clk_src_t rmt_clock_source_t
   RMT group clock source.

Note: User should select the clock source based on the power and resolution requirement

2.5.17  SD SPI Host Driver

Overview

The SD SPI host driver allows communication with one or more SD cards using the SPI Master driver, which utilizes the SPI host. Each card is accessed through an SD SPI device, represented by an SD SPI handle `sdspi_dev_handle_t`, which returns when the device is attached to an SPI bus by calling `sdspi_host_init_device()`. It is important to note that the SPI bus should be initialized beforehand by `spi_bus_initialize()`.

With the help of SPI Master Driver the SD SPI host driver based on, the SPI bus can be shared among SD cards and other SPI devices. The SPI Master driver will handle exclusive access from different tasks.

The SD SPI driver uses software-controlled CS signal.

How to Use

Firstly, use the macro `SDSPI_DEVICE_CONFIG_DEFAULT` to initialize the structure `sdspi_device_config_t`, which is used to initialize an SD SPI device. This macro will also fill in the default pin mappings, which are the same as the pin mappings of the SDMMC host driver. Modify the host and pins of the structure to desired value. Then call `sdspi_host_init_device` to initialize the SD SPI device and attach to its bus.
Then use the `SDSPI_HOST_DEFAULT` macro to initialize the `sdmmc_host_t` structure, which is used to store the state and configurations of the upper layer (SD/SDIO/MMC driver). Modify the slot parameter of the structure to the SD SPI device SD SPI handle just returned from `sdspi_host_init_device`. Call `sdspi_card_init` with the `sdmmc_host_t` to probe and initialize the SD card.

Now you can use SD/SDIO/MMC driver functions to access your card!

**Other Details**

Only the following driver’s API functions are normally used by most applications:

- `sdspi_host_init()`
- `sdspi_host_init_device()`
- `sdspi_host_remove_device()`
- `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 `SD/SDIO/MMC Driver`.

**Note:** SD over SPI does not support speeds above `SDMMC_FREQ_DEFAULT` due to the limitations of the SPI driver.

---

**Warning:** If you want to share the SPI bus among SD card and other SPI devices, there are some restrictions, see `Sharing the SPI Bus Among SD Cards and Other SPI Devices`.

---

**Related Docs**

**Sharing the SPI Bus Among SD Cards and Other SPI Devices**

The SD card has an SPI mode, enabling it to function as an SPI device, but there are some restrictions that we need to pay attention to.

**Pin Loading of Other Devices** When adding more devices onto the same bus, the overall pin loading increases. The loading consists of AC loading (pin capacitor) and DC loading (pull-ups).

**AC Loading** SD cards, designed for high-speed communications, have small pin capacitors (AC loading) to work until 50 MHz. However, the other attached devices will increase the pin’s AC loading.

Heavy AC loading of a pin may prevent the pin from being toggled quickly. By using an oscilloscope, you will see the edges of the pin become smoother, i.e., the gradient of the edge is smaller. The setup timing requirements of an SD card may be violated when the card is connected to a bus with a high AC load. Even worse, high AC loads may cause the SD card and other SPI devices to fail to properly resolve clock signals from the host, affecting communication stability.

This issue may be more obvious if other attached devices are not designed to work at the same frequency as the SD card, because they may have larger pin capacitors. The larger the pin capacity, the greater the pin response time, the smaller the max frequency the SD bus can work.

To see if your pin AC loading is too heavy, you can try the following tests:

**Terminology:**

- **launch edge**: at which clock edge the data starts to toggle;
- **latch edge**: at which clock edge the data is supposed to be sampled by the receiver. For SD card, it is the rising edge.

1. Use an oscilloscope to see the clock and compare the data line to the clock.
• If you see the clock is not fast enough, e.g., the rising/falling edge is longer than 1/4 of the clock cycle, it means the clock is skewed too much.
• If you see the data line unstable before the latch edge of the clock, it means the load of the data line is too large.

You may also observe the corresponding phenomenon that data delayed largely from the launching edge of the clock with logic analyzers. But it is not as obvious as with an oscilloscope.

2. Try to use a slower clock frequency.
   If the lower frequency can work while the higher frequency cannot, it is an indication that the AC loading on the pins is too large.

If the AC loading of the pins is too large, you can either use other faster devices with lower pin load or slow down the clock speed.

DC Loading
The pull-ups required by SD cards are usually around 10 kOhm to 50 kOhm, which may be too strong for some other SPI devices.

Check the specification of your device about its DC output current, it should be larger than 700 μA, otherwise, the device output may not be read correctly.

Initialization Sequence
Note: If you see any problem in the following steps, please make sure the timing is correct first. You can try to slow down the clock speed, such as setting SDMMC_FREQ_PROBING to 400 kHz for SD card, to avoid the influence of pin AC loading, as discussed in the previous section.

When using an SD card with other SPI devices on the same SPI bus, due to the restrictions of the SD card startup flow, the following initialization sequence should be followed. Refer to storage/sd_card for further details.

1. Initialize the SPI bus properly by `spi_bus_initialize()`.
2. Tie the CS lines of all other devices than the SD card to idle state (by default it’s high). This is to avoid conflicts with the SD card in the following step.
   You can do this by either:
   1. Attach devices to the SPI bus by calling `spi_bus_add_device()`. This function will by default initialize the GPIO that is used as CS to the idle level: high.
   2. Initialize GPIO on the CS pin that needs to be tied up before actually adding a new device.
   3. Rely on the internal/external pull-up (not recommended) to pull up all the CS pins when the GPIOs of ESP are not initialized yet. You need to check carefully the pull-up is strong enough and there are no other pull-downs that will influence the pull-up. For example, internal pull-down should be enabled.
3. Mount the card to the filesystem by calling `esp_vfs_fat_sdspi_mount()`. This step will put the SD card into the SPI mode, which should be done before all other SPI communications on the same bus. Otherwise, the card will stay in the SD mode, in which mode it may randomly respond to any SPI communications on the bus, even when its CS line is not addressed.
   If you want to test this behavior, please also note that, once the card is put into SPI mode, it will not return to SD mode before the next power cycle, i.e., powered down and powered up again.
4. Now you can talk to other SPI devices freely!

API Reference

Header File

• components/driver/sdspi/include/driver/sdspi_host.h
• This header file can be included with:

  ```
  #include "driver/sdspi_host.h"
  ```

• This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
API Reference

Chapter 2

REQUIRES driver

or

PRIV_REQUIRES driver

Functions

**esp_err_t sdspi_host_init**(void)

Initialize SD SPI driver.

**Note:** This function is not thread safe

**Returns**

- ESP_OK on success
- other error codes may be returned in future versions

**esp_err_t sdspi_host_init_device**(const sdspi_device_config_t *dev_config, sdspi_dev_handle_t *out_handle)

Attach and initialize an SD SPI device on the specific SPI bus.

**Note:** This function is not thread safe

**Note:** Initialize the SPI bus by *spi_bus_initialize()* before calling this function.

**Note:** The SDIO over sdspi needs an extra interrupt line. Call *gpio_install_isr_service()* before this function.

**Parameters**

- **dev_config** - pointer to device configuration structure
- **out_handle** - Output of the handle to the sdspi device.

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_ARG if sdspi_host_init_device has invalid arguments
- ESP_ERR_NO_MEM if memory can not be allocated
- other errors from the underlying spi_master and gpio drivers

**esp_err_t sdspi_host_remove_device**(sdspi_dev_handle_t handle)

Remove an SD SPI device.

**Parameters**

- **handle** - Handle of the SD SPI device

**Returns**

- Always ESP_OK

**esp_err_t sdspi_host_do_transaction**(sdspi_dev_handle_t handle, 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.
**Parameters**
- `handle` - Handle of the sdspi device
- `cmdinfo` - pointer to structure describing command and data to transfer

**Returns**
- 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

```c
esp_err_t sdspi_host_set_card_clk (sdspi_dev_handle_t host, 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

**Parameters**
- `host` - Handle of the sdspi device
- `freq_khz` - card clock frequency, in kHz

**Returns**
- ESP_OK on success
- other error codes may be returned in the future

```c
esp_err_t sdspi_host_get_real_freq (sdspi_dev_handle_t handle, int *real_freq_khz)
```

Calculate working frequency for specific device.

**Parameters**
- `handle` - SDspi device handle
- `real_freq_khz` - [out] output parameter to hold the calculated frequency (in kHz)

**Returns**
- ESP_ERR_INVALID_ARG : `handle` is NULL or invalid or `real_freq_khz` parameter is NULL
- ESP_OK : Success

```c
esp_err_t sdspi_host_deinit (void)
```

Release resources allocated using sdspi_host_init.

**Note:** This function is not thread safe

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_STATE if sdspi_host_init function has not been called

```c
esp_err_t sdspi_host_io_int_enable (sdspi_dev_handle_t handle)
```

Enable SDIO interrupt.

**Parameters**
- `handle` - Handle of the sdspi device

**Returns**
- ESP_OK on success

```c
esp_err_t sdspi_host_io_int_wait (sdspi_dev_handle_t handle, TickType_t timeout_ticks)
```

Wait for SDIO interrupt until timeout.

**Parameters**
- `handle` - Handle of the sdspi device
- `timeout_ticks` - Ticks to wait before timeout.
Returns

- ESP_OK on success

Structures

struct sdspi_device_config_t
Extra configuration for SD SPI device.

Public Members

spi_host_device_t host_id
SPI host to use, SPIx_HOST (see spi_types.h).

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.

bool gpio_wp_polarity
GPIO write protect polarity 0 means “active low”, i.e. card is protected when the GPIO is low; 1 means “active high”, i.e. card is protected when GPIO is high.

Macros

SDSPI_DEFAULT_HOST
SDSPI_DEFAULT_DMA

SDSPI_HOST_DEFAULT()
Default sdmmc_host_t structure initializer for SD over SPI driver.
Uses SPI mode and max frequency set to 20MHz
‘slot’ should be set to an sdspi device initialized by sdspi_host_init_device().

SDSPI_SLOT_NO_CS
indicates that card select line is not used

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_IO_ACTIVE_LOW

SDSPI_DEVICE_CONFIG_DEFAULT()
Macro defining default configuration of SD SPI device.

Type Definitions
typedef int sdspi_dev_handle_t
Handle representing an SD SPI device.

2.5.18 Sigma-Delta Modulation (SDM)

Introduction
ESP32-S2 has a second-order sigma-delta modulator, which can generate independent PDM pulses to multiple channels. Please refer to the TRM to check how many hardware channels are available.\(^1\)

Delta-sigma modulation converts an analog voltage signal into a pulse frequency, or pulse density, which can be understood as pulse-density modulation (PDM) (refer to Delta-sigma modulation on Wikipedia).

The main differences comparing to I2S PDM mode and DAC peripheral are:

1. SDM has no clock signal, it is just like the DAC mode of PDM;
2. SDM has no DMA, and it can not change its output density continuously. If you have to, you can update the density in a timer’s callback;
3. Based on the former two points, unlike the DAC peripheral, an external active or passive low-pass filter is required additionally to restore the analog wave (See Convert to an Analog Signal (Optional)).

Typically, a Sigma-Delta modulated channel can be used in scenarios like:

- LED dimming
- Simple DAC (8-bit), with the help of an active RC low-pass filter
- Class D amplifier, with the help of a half-bridge or full-bridge circuit plus an LC low-pass filter

Functional Overview
The following sections of this document cover the typical steps to install and operate an SDM channel:

- **Resource Allocation** - covers how to initialize and configure an SDM channel and how to recycle the resources when it finishes working.
- **Enable and Disable Channel** - covers how to enable and disable the channel.
- **Set Pulse Density** - describes how to set the equivalent duty cycle of the PDM pulses.
- **Power Management** - describes how different source clock selections can affect power consumption.
- **IRAM Safe** - lists which functions are supposed to work even when the cache is disabled.
- **Thread Safety** - lists which APIs are guaranteed to be thread-safe by the driver.
- **Kconfig Options** - lists the supported Kconfig options that can be used to make a different effect on driver behavior.

\(^1\) Different ESP chip series might have different numbers of SDM channels. Please refer to Chapter GPIO and IOMUX in ESP32-S2 Technical Reference Manual for more details. The driver does not forbid you from applying for more channels, but it will return an error when all available hardware resources are used up. Please always check the return value when doing resource allocation (e.g., `sdm_new_channel()`).
**Resource Allocation**  In ESP-IDF, the information and attributes of SDM channels are managed and accessed through specific data structures, where the data structure is called `sdm_channel_handle_t`. Each channel is capable to output the binary, hardware-generated signal with the sigma-delta modulation. The driver manages all available channels in a pool so that there is no need to manually assign a fixed channel to a GPIO.

To install an SDM channel, you should call `sdm_new_channel()` to get a channel handle. Channel-specific configurations are passed in the `sdm_config_t` structure:

- `sdm_config_t::gpio_num` sets the GPIO that the PDM pulses output from.
- `sdm_config_t::clk_src` selects the source clock for the SDM module. Note that, all channels should select the same clock source.
- `sdm_config_t::sample_rate_hz` sets the sample rate of the SDM module.
- `sdm_config_t::invert_out` sets whether to invert the output signal.
- `sdm_config_t::io_loop_back` is for debugging purposes only. It enables both the GPIO’s input and output ability through the GPIO matrix peripheral.

The function `sdm_new_channel()` can fail due to various errors such as insufficient memory, invalid arguments, etc. Specifically, when there are no more free channels (i.e., all hardware SDM channels have been used up), `ESP_ERR_NOT_FOUND` will be returned.

If a previously created SDM channel is no longer required, you should recycle it by calling `sdm_del_channel()`. It allows the underlying HW channel to be used for other purposes. Before deleting an SDM channel handle, you should disable it by `sdm_channel_disable()` in advance or make sure it has not been enabled yet by `sdm_channel_enable()`.

### Creating an SDM Channel with a Sample Rate of 1 MHz

```c
sdm_channel_handle_t chan = NULL;
sdm_config_t config = {
   .clk_src = SDM_CLK_SRC_DEFAULT,
   .sample_rate_hz = 1 * 1000 * 1000,
   .gpio_num = 0,
};
ESP_ERROR_CHECK(sdm_new_channel(&config, &chan));
```

### Enable and Disable Channel

Before doing further IO control to the SDM channel, you should enable it first, by calling `sdm_channel_enable()` . Internally, this function:

- switches the channel state from `init` to `enable`
- acquires a proper power management lock if a specific clock source (e.g., APB clock) is selected. See also Power Management for more information.

On the contrary, calling `sdm_channel_disable()` does the opposite, that is, put the channel back to the `init` state and releases the power management lock.

### Set Pulse Density

For the output PDM signals, the pulse density decides the output analog voltage that is restored by a low-pass filter. The restored analog voltage from the channel is calculated by 

\[
V_{out} = \frac{V_{DD, IO}}{256} \times \text{duty} + \frac{V_{DD, IO}}{2}
\]

The range of the quantized density input parameter of `sdm_channel_set_pulse_density()` is from -128 to 127 (8-bit signed integer). Depending on the value of the density parameter, the duty cycle of the output signal will be changed accordingly. For example, if a zero value is set, then the output signal’s duty will be around 50%.

### Power Management

When power management is enabled (i.e., `CONFIG_PM_ENABLE` is on), the system will adjust the APB frequency before going into Light-sleep, thus potentially changing the sample rate of the sigma-delta modulator.

However, the driver can prevent the system from changing APB frequency by acquiring a power management lock of type `ESP_PM_APB_FREQ_MAX`. Whenever the driver creates an SDM channel instance that has selected `SDM_CLK_SRC_APB` as its clock source, the driver guarantees that the power management lock is ac-
quired when enabling the channel by `sdm_channel_enable()`. Likewise, the driver releases the lock when `sdm_channel_disable()` is called for that channel.

**IRAM Safe** There is a Kconfig option `CONFIG_SDM_CTRL_FUNC_IN_IRAM` that can put commonly-used IO control functions into IRAM as well. So that these functions can also be executable when the cache is disabled. These IO control functions are listed as follows:

- `sdm_channel_set_pulse_density()`

**Thread Safety** The factory function `sdm_new_channel()` is guaranteed to be thread-safe by the driver, which means, the user can call it from different RTOS tasks without protection by extra locks.

The following functions are allowed to run under ISR context, the driver uses a critical section to prevent them being called concurrently in both task and ISR.

- `sdm_channel_set_pulse_density()`

Other functions that take the `sdm_channel_handle_t` as the first positional parameter, are not treated as thread-safe. This means the user should avoid calling them from multiple tasks.

**Kconfig Options**

- `CONFIG_SDM_CTRL_FUNC_IN_IRAM` controls where to place the SDM channel control functions (IRAM or Flash), see **IRAM Safe** for more information.
- `CONFIG_SDM_ENABLE_DEBUG_LOG` is used to enable the debug log output. Enabling this option increases the firmware binary size.

**Convert to an Analog Signal (Optional)**

Typically, if a Sigma-Delta signal is connected to an LED to adjust the brightness, you do not have to add any filter between them, because our eyes have their own low-pass filters for changes in light intensity. However, if you want to check the real voltage or watch the analog waveform, you need to design an analog low-pass filter. Also, it is recommended to use an active filter instead of a passive filter to gain better isolation and not lose too much voltage.

For example, you can take the following **Sallen-Key topology Low Pass Filter** as a reference.

**Application Example**

- 100 Hz sine wave that is modulated with Sigma-Delta: `peripherals/sigma_delta/sdm_dac`
- LED driven by a GPIO that is modulated with Sigma-Delta: `peripherals/sigma_delta/sdm_led`

**API Reference**

**Header File**

- `components/driver/sigma_delta/include/driver/sdm.h`
- This header file can be included with:

```c
#include "driver/sdm.h"
```

- This header file is a part of the API provided by the `driver` component. To declare that your component depends on `driver`, add the following to your `CMakeLists.txt`:

```cmake
REQUIRES driver
```

or

```cmake
PRIV_REQUIRES driver
```
Chapter 2. API Reference

Fig. 20: Sallen-Key Low Pass Filter

Functions

esp_err_t sdm_new_channel (const sdm_config_t *config, sdm_channel_handle_t *ret_chan)

Create a new Sigma Delta channel.

Parameters

• config - [in] SDM configuration
• ret_chan - [out] Returned SDM channel handle

Returns

• ESP_OK: Create SDM channel successfully
• ESP_ERR_INVALID_ARG: Create SDM channel failed because of invalid argument
• ESP_ERR_NO_MEM: Create SDM channel failed because out of memory
• ESP_ERR_NOT_FOUND: Create SDM channel failed because all channels are used up and no more free one
• ESP_FAIL: Create SDM channel failed because of other error

esp_err_t sdm_del_channel (sdm_channel_handle_t chan)

Delete the Sigma Delta channel.

Parameters chan - [in] SDM channel created by sdm_new_channel

Returns

• ESP_OK: Delete the SDM channel successfully
• ESP_ERR_INVALID_ARG: Delete the SDM channel failed because of invalid argument
• ESP_ERR_INVALID_STATE: Delete the SDM channel failed because the channel is not in init state
• ESP_FAIL: Delete the SDM channel failed because of other error

esp_err_t sdm_channel_enable (sdm_channel_handle_t chan)

Enable the Sigma Delta channel.

Note: This function will transit the channel state from init to enable.

Note: This function will acquire a PM lock, if a specific source clock (e.g. APB) is selected in the
Chapter 2. API Reference

.sdmcfg_, while CONFIG_PM_ENABLE is enabled.

Parameters chan - [in] SDM channel created by sdm_new_channel

Returns

- ESP_OK: Enable SDM channel successfully
- ESP_ERR_INVALID_ARG: Enable SDM channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Enable SDM channel failed because the channel is already enabled
- ESP_FAIL: Enable SDM channel failed because of other error

.esp.err_.sdm_channel_disable(sdm_channel_handle_t chan)

Disable the Sigma Delta channel.

Note: This function will do the opposite work to the sdm_channel_enable()

Parameters chan - [in] SDM channel created by sdm_new_channel

Returns

- ESP_OK: Disable SDM channel successfully
- ESP_ERR_INVALID_ARG: Disable SDM channel failed because of invalid argument
- ESP_ERR_INVALID_STATE: Disable SDM channel failed because the channel is not enabled yet
- ESP_FAIL: Disable SDM channel failed because of other error

.esp.err_.sdm_channel_set_pulse_density(sdm_channel_handle_t chan, int8_t density)

Set the pulse density of the PDM output signal.

Note: The raw output signal requires a low-pass filter to restore it into analog voltage, the restored analog output voltage could be Vout = VDD_IO / 256 * density + VDD_IO / 2

Note: This function is allowed to run within ISR context

Note: This function will be placed into IRAM if CONFIG_SDM_CTRL_FUNC_IN_IRAM is on, so that it’s allowed to be executed when Cache is disabled

Parameters

- chan - [in] SDM channel created by sdm_new_channel
- density - [in] Quantized pulse density of the PDM output signal, ranges from -128 to 127. But the range of [-90, 90] can provide a better randomness.

Returns

- ESP_OK: Set pulse density successfully
- ESP_ERR_INVALID_ARG: Set pulse density failed because of invalid argument
- ESP_FAIL: Set pulse density failed because of other error

.esp.err_.sdm_channel_set_duty(sdm_channel_handle_t chan, int8_t duty)

The alias function of sdm_channel_set_pulse_density, it decides the pulse density of the output signal.

Note: sdm_channel_set_pulse_density has a more appropriate name compare this alias function, suggest to turn to sdm_channel_set_pulse_density instead
Parameters

• chan - [in] SDM channel created by sdm_new_channel
• duty - [in] Actually it’s the quantized pulse density of the PDM output signal

Returns

• ESP_OK: Set duty cycle successfully
• ESP_ERR_INVALID_ARG: Set duty cycle failed because of invalid argument
• ESP_FAIL: Set duty cycle failed because of other error

Structures

struct sdm_config_t
    Sigma Delta channel configuration.

Public Members

int gpio_num
    GPIO number

sdm_clock_source_t clk_src
    Clock source

uint32_t sample_rate_hz
    Over sample rate in Hz, it determines the frequency of the carrier pulses

uint32_t invert_out
    Whether to invert the output signal

uint32_t io_loop_back
    For debug/test, the signal output from the GPIO will be fed to the input path as well

struct sdm_config_t::[anonymous] flags
    Extra flags

Type Definitions

typedef struct sdm_channel_t *sdm_channel_handle_t
    Type of Sigma Delta channel handle.

Header File

• components/hal/include/hal/sdm_types.h
• This header file can be included with:

```
#include "hal/sdm_types.h"
```

Type Definitions

typedef soc_periph_sdm_clk_src_t sdm_clock_source_t
2.5.19 SPI Flash API

Overview

The spi_flash component contains API functions related to reading, writing, erasing, and memory mapping for data in the external flash.

For higher-level API functions which work with partitions defined in the partition table, see Partitions API

Note: esp_partition_* APIs are recommended to be used instead of the lower level esp_flash_* API functions when accessing the main SPI flash chip, since they conduct bounds checking and are guaranteed to calculate correct offsets in flash based on the information in the partition table. esp_flash_* functions can still be used directly when accessing an external (secondary) SPI flash chip.

Different from the API before ESP-IDF v4.0, the functionality of esp_flash_* APIs 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 external flash chips connected to not only SPI0/1 but also other SPI buses like SPI2.

Note: Instead of going through the cache connected to the SPI0 peripheral, most esp_flash_* APIs go through other SPI peripherals like SPI1, SPI2, etc. This makes them able to access not only the main flash, but also external (secondary) flash.

However, due to the limitations of the cache, operations through the cache are limited to the main flash. The address range limitation for these operations is also on the cache side. The cache is not able to access external flash chips or address range above its capabilities. These cache operations include: mmap, encrypted read/write, executing code or access to variables in the flash.

Note: Flash APIs after ESP-IDF v4.0 are no longer atomic. If a write operation occurs during another on-going read operation, and the flash addresses of both operations overlap, the data returned from the read operation may contain both old data and new data (that was updated written by the write operation).

Note: Encrypted flash operations are only supported with the main flash chip (and not with other flash chips, that is on SPI1 with different CS, or on other SPI buses). Reading through cache is only supported on the main flash, which is determined by the HW.

Support for Features of Flash Chips

Quad/Dual Mode Chips Features of different flashes are implemented in different ways and thus need special support. The fast/slow read and Dual mode (DOUT/DIO) of almost all flashes with 24-bit address are supported, because they do not need any vendor-specific commands.

Quad mode (QIO/QOUT) is supported on the following chip types:

1. ISSI
2. GD
3. MXIC
4. FM
5. Winbond
6. XMC
7. BOYA
Note: Only when one flash series listed above is supported by ESP32-S2, this flash series is supported by the chip driver by default. You can use `Component config > SPI Flash driver > Auto-detect flash chips` in `menuconfig` to enable/disable a flash series.

Optional Features

Optional Features for Flash Some features are not supported on all ESP chips and Flash chips. You can check the list below for more information.

- Auto Suspend & Resume
- Flash unique ID
- High performance mode
- OPI flash support
- 32-bit Address Flash Chips

Note: When Flash optional features listed in this page are used, aside from the capability of ESP chips, and ESP-IDF version you are using, you will also need to make sure these features are supported by flash chips used.

- If you are using an official Espressif modules/SiP, Some of the modules/SiPs always support the feature, in this case you can see these features listed in the datasheet. Otherwise please contact Espressif’s business team to know if we can supply such products for you.
- If you are making your own modules with your own bought flash chips, and you need features listed above. Please contact your vendor if they support the those features, and make sure that the chips can be supplied continuously.

Attention: This document only shows that ESP-IDF code has supported the features of those flash chips. It is not a list of stable flash chips certified by Espressif. If you build your own hardware from flash chips with your own bought flash chips (even with flash listed in this page), you need to validate the reliability of flash chips yourself.

Auto Suspend & Resume This feature is only supported on ESP32-S3, ESP32-C2, ESP32-C3, ESP32-C6, ESP32-H2 for now.

The support for ESP32-P4 may be added in the future.

Flash Unique ID This feature is supported on all Espressif chips.

Unique ID is not flash id, which means flash has 64-Bit unique ID for each device. The instruction to read the unique ID (4Bh) accesses a factory-set read-only 64-bit number that is unique to each flash device. This ID number helps you to recognize each single device. Not all flash vendors support this feature. If you try to read the unique ID on a chip which does not have this feature, the behavior is not determined. The support list is as follows.

List of Flash chips that support this feature:

1. ISSI
2. GD
3. TH
4. FM
5. Winbond
6. XMC
7. BOYA
**High Performance Mode**  This feature is only supported on ESP32-S3 for now.

The support for ESP32-S2, ESP32-C3, ESP32-C6, ESP32-H2, ESP32-P4 may be added in the future.

**OPI flash Support**  This feature is only supported on ESP32-S3 for now.

OPI flash means that the flash chip supports octal peripheral interface, which has octal I/O pins. Different octal flash has different configurations and different commands. Hence, it is necessary to carefully check the support list.

**32-bit Address Flash Chips**  This feature is supported on all Espressif chips (with various restrictions to applications).

Most NOR flash chips used by Espressif chips use 24-bits address, which can cover 16 MBytes memory. However, for larger memory (usually equal to or larger than 16 MBytes), flash uses a 32-bits address to address larger memory. Regretfully, 32-bits address chips have vendor-specific commands, so we need to support the chips one by one.

List of Flash chips that support this feature:

1. W25Q256
2. GD25Q256

**Important:** Over 16 MBytes space on flash mentioned above can only be used for data saving, like file system. If your data/instructions over 16 MBytes spaces need to be mapped to MMU (so as to be accessed by the CPU), please enable the config `IDF_EXPERIMENTAL_FEATURES` and `BOOTLOADER_CACHE_32BIT_ADDR_FLASH` and read the limitations following:

1. This feature is valid only for 4-line flash. Octal flash supports 32-bit-addr by default
2. This feature needs the MMU on ESP chip to be able to map to >= 16 MB physical address on the Flash. (Only ESP32S3 supports this up to now)
3. This option is experimental, which means it can not use on all flash chips stable, for more information, please contact Espressif Business support.

There are some features that are not supported by all flash chips, or not supported by all Espressif chips. These features include:

- 32-bit address flash - usually means that the flash has higher capacity (equal to or larger than 16 MB) that needs longer addresses.
- Flash unique ID - means that flash supports its unique 64-bit ID.

If you want to use these features, please ensure both ESP32-S2 and ALL flash chips in your product support these features. For more details, refer to Optional Features for Flash.

You may also customise your own flash chip driver. See Overriding Default Chip Drivers for more details.

**Warning:** Customizing SPI Flash Chip Drivers is considered an “expert” feature. Users should only do so at their own risk. (See the notes below)

**Overriding Default Chip Drivers**  During the SPI Flash driver’s initialization (i.e., `esp_flash_init()`), there is a chip detection step during which the driver iterates through a Default Chip Driver List and determine which chip driver can properly support the currently connected flash chip. The Default Chip Drivers are provided by the ESP-IDF, thus are updated in together with each ESP-IDF version. However ESP-IDF also allows users to customize their own chip drivers.

Users should note the following when customizing chip drivers:

1. You may need to rely on some non-public ESP-IDF functions, which have slight possibility to change between ESP-IDF versions. On the one hand, these changes may be useful bug fixes for your driver, on the other hand, they may also be breaking changes (i.e., breaks your code).
2. Some ESP-IDF bug fixes to other chip drivers are not automatically applied to your own custom chip drivers.
3. If the protection of flash is not handled properly, there may be some random reliability issues.
4. If you update to a newer ESP-IDF version that has support for more chips, you will have to manually add those new chip drivers into your custom chip driver list. Otherwise the driver will only search for the drivers in custom list you provided.

Steps For Creating Custom Chip Drivers and Overriding the ESP-IDF Default Driver List

1. Enable the `CONFIG_SPI_FLASH_OVERRIDE_CHIP_DRIVER_LIST` config option. This prevents compilation and linking of the Default Chip Driver List (default_registered_chips) provided by ESP-IDF. Instead, the linker searches for the structure of the same name (default_registered_chips) that must be provided by the user.
2. Add a new component in your project, e.g., `custom_chip_driver`.
3. Copy the necessary chip driver files from the `spi_flash` component in ESP-IDF. This may include:
   - `spi_flash_chip_drivers.c` (to provide the default_registered_chips structure)
   - Any of the `spi_flash_chip_*.c` files that matches your own flash model best
   - `CMakeLists.txt` and `linker.lf` files

Modify the files above properly. Including:
   - Change the default_registered_chips variable to non-static and remove the #ifdef logic around it.
   - Update linker.lf file to rename the fragment header and the library name to match the new component.
   - If reusing other drivers, some header names need prefixing with `spi_flash/` when included from outside `spi_flash` component.

Note:

- When writing your own flash chip driver, you can set your flash chip capabilities through `spi_flash_chip_***`(vendor)_get_caps and points the function pointer `get_chip_caps` for protection to the `spi_flash_chip_***_get_caps` function. The steps are as follows.
  1. Please check whether your flash chip have the capabilities listed in `spi_flash_caps_t` by checking the flash datasheet.
  2. Write a function named `spi_flash_chip_***`(vendor)_get_caps. Take the example below as a reference. (if the flash support suspend and read unique id).
  3. Points the pointer `get_chip_caps` (in `spi_flash_chip_t`) to the function mentioned above.

```c
spi_flash_caps_t spi_flash_chip_***`(vendor)_get_caps(esp_flash_t *chip)
{
    spi_flash_caps_t caps_flags = 0;
    // 32-bit-address flash is not supported
    flash_suspend is supported
    caps_flags |= SPI_FLAHS_CHIP_CAP_SUSPEND;
    // flash read unique id.
    caps_flags |= SPI_FLASH_CHIP_CAP_UNIQUE_ID;
    return caps_flags;
}
```

```c
const spi_flash_chip_t esp_flash_chip_eon = {
    // Other function pointers
    .get_chip_caps = spi_flash_chip_eon_get_caps,
};
```

- You also can see how to implement this in the example `storage/custom_flash_driver`.

4. Write a new `CMakeLists.txt` file for the `custom_chip_driver` component, including an additional line to add a linker dependency from `spi_flash` to `custom_chip_driver`:
Chapter 2. API Reference

idf_component_register(SRCS "spi_flash_chip_drivers.c"
    "spi_flash_chip_mychip.c"  # modify as needed
    REQUIRES hal
    PRIV_REQUIRES spi_flash
    LDFRAGMENTS linker.lf)
idf_component_add_link_dependency(FROM spi_flash)

• An example of this component CMakeLists.txt can be found in storage/custom_flash_driver/components/custom_chip_driver/CMakeLists.txt

5. The linker.lf is used to put every chip driver that you are going to use whilst cache is disabled into internal RAM. See Linker Script Generation for more details. Make sure this file covers all the source files that you add.

6. Build your project, and you will see the new flash driver is used.

Example  See also storage/custom_flash_driver.

Initializing a Flash Device

To use the esp_flash_* APIs, you need to initialise a flash chip on a certain SPI bus, as shown below:

1. Call `spi_bus_initialize()` to properly initialize an SPI bus. This function initializes 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 to the bus. This function allocates memory and fills 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 also detects the chip type, and influence the following operations.

Note: Multiple flash chips can be attached to the same bus now.

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_ESP TOOLPY_FLASHSIZE` in the 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

The SPI0/1 bus is shared between the instruction & data cache (for firmware execution) and the SPI1 peripheral (controlled by the drivers including this SPI Flash driver). Hence, operations to SPI1 will cause significant influence to the whole system. This kind of operations include calling SPI Flash API or other drivers on SPI1 bus, any operations like read/write/erase or other user defined SPI operations, regardless to the main flash or other SPI slave devices.

On ESP32-S2, the config options `CONFIG_SPIRAM_FETCH_INSTRUCTIONS` (disabled by default) and `CONFIG_SPIRAM_RODATA` (disabled by default) allow the cache to read/write PSRAM concurrently with SPI1 operations. See XIP from PSRAM Feature for more details.

If these options are disabled, the caches must be disabled while reading/writing/erasing operations. There are some constraints using driver on the SPI1 bus, see When the Caches Are Disabled. These constraints will cause more IRAM/DRAM usages.

When the Caches Are Disabled Under this condition, all CPUs should always execute code and access data from internal RAM. The APIs documented in this file will disable the caches automatically and transparently.

**Note:** When `CONFIG_SPIRAM_FETCH_INSTRUCTIONS` and `CONFIG_SPIRAM_RODATA` are both enabled, these APIs will not disable the caches.

The way that these APIs disable the caches also disables non-IRAM-safe interrupts. These will be restored until the Flash operation completes.

See also OS Functions and SPI Bus Lock.

There are no such constraints and impacts for flash chips on other SPI buses than SPI0/1.

For differences between internal RAM (e.g., IRAM, DRAM) and flash cache, please refer to the application memory layout documentation.

**IRAM-Safe Interrupt Handlers** For interrupt handlers which need to execute when the cache is disabled (e.g., 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. See How to Place Code in IRAM.

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).

**Note:** When working with strings in ISRs, it is not advised to use printf and other output functions. For debugging purposes, use `ESP_DRAM_LOGE()` and similar macros when logging from ISRs. Make sure that both TAG and format string are placed into DRAM in that case.

**Non-IRAM-Safe Interrupt Handlers** If the `ESP_INTR_FLAG_IRAM` flag is not set when registering, the interrupt handler will not get executed when the caches are disabled. Once the caches are restored, the non-IRAM-safe interrupts will be re-enabled. After this moment, the interrupt handler will run normally again. This means that as long as caches are disabled, users will not see the corresponding hardware event happening.

**XIP from PSRAM Feature** If `CONFIG_SPIRAM_FETCH_INSTRUCTIONS` is enabled, the flash .text sections (used for instructions) will be placed in PSRAM.

If `CONFIG_SPIRAM_RODATA` is enabled, the flash .rodata sections (used for read only data) will be placed in PSRAM.
Chapter 2. API Reference

The corresponding virtual memory range will be re-mapped to PSRAM.

If both of the above options are enabled, the Cache won’t be disabled during an SPI1 Flash operation. You don’t need to make sure ISRs, ISR callbacks and involved data are placed in internal RAM.

**Attention:** The SPI0/1 bus is shared between the instruction & data cache (for firmware execution) and the SPI1 peripheral (controlled by the drivers including this SPI flash driver). Hence, calling SPI Flash API on SPI1 bus (including the main flash) causes significant influence to the whole system. See *Concurrency Constraints for Flash on SPI1* for more details.

### 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-S2 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 64 KB pages. Memory mapping hardware can map flash into the data address space and the instruction address space. See the technical reference manual for more details and limitations about memory mapping hardware.

Note that some pages are used to map the application itself into memory, so the actual number of available pages may be less than the capability of the hardware.

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 `spi_flash_mmap.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 64 KB aligned physical address.
- `esp_partition_mmap()` may be given any arbitrary offset within the partition. It adjusts the returned pointer to mapped memory as necessary.

Note that since memory mapping happens in pages, it may be possible to read data outside of the partition provided to `esp_partition_mmap`, regardless of the partition boundary.

**Note:** mmap is supported by cache, so it can only be used on main flash.

### SPI Flash 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, provide support of some OS functions (e.g., lock, delay) in different stages (1st/2nd boot, or the app).
### Host Driver

The host driver relies on an interface (`spi_flash_host_driver_t`) defined in the `spi_flash_types.h` (in the hal/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-S2 memory-spi functionalities. However, due to the speed limitations of ESP32-S2, the HAL layer cannot provide high-speed implementations to some reading commands (so the support for it was dropped). 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 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 afterwards. Some chips need different commands or values, 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 entries of a lock and delay.

The lock (see SPI Bus Lock) is used to resolve the conflicts among the access of devices on the same SPI bus, and the SPI Flash chip access. E.g.

1. On SPI1 bus, the cache (used to fetch the data (code) in the Flash and PSRAM) should be disabled when the flash chip on the SPI0/1 is being accessed.
2. On the other buses, the flash driver needs to disable the ISR registered by SPI Master driver, to avoid conflicts.
3. Some devices of SPI Master driver may require to use the bus monopolized during a period (especially when the device does not have a CS wire, or the wire is controlled by software like SDSPi driver).

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.

OS functions can also help to avoid a watchdog timeout when erasing large flash areas. During this time, the CPU is occupied with the flash erasing task. This stops other tasks from being executed. Among these tasks is the idle task to feed the watchdog timer (WDT). If the configuration option `CONFIG_ESP_TASK_WDT_PANIC` is selected and the flash operation time is longer than the watchdog timeout period, the system will reboot.

It is pretty hard to totally eliminate this risk, because the erasing time varies with different flash chips, making it hard to be compatible in flash drivers. Therefore, users need to pay attention to it. Please use the following guidelines:

1. It is recommended to enable the `CONFIG_SPI_FLASH_YIELD_DURING_ERASE` option to allow the scheduler to re-schedule during erasing flash memory. Besides, following parameters can also be used.
   - Increase `CONFIG_SPI_FLASH_ERASE_YIELD_TICKS` or decrease `CONFIG_SPI_FLASH_ERASE_YIELD_DURATION_MS` in menuconfig.
   - You can also increase `CONFIG_ESP_TASK_WDT_TIMEOUT_S` in menuconfig for a larger watchdog timeout period. However, with larger watchdog timeout period, previously detected timeouts may no longer be detected.

2. Please be aware of the consequences of enabling the `CONFIG_ESP_TASK_WDT_PANIC` option when doing long-running SPI flash operations which triggers the panic handler when it times out. However, this option can also help dealing with unexpected exceptions in your application. Please decide whether this is needed to be enabled according to actual condition.

3. During your development, please carefully review the actual flash operation according to the specific requirements and time limits on erasing flash memory of your projects. Always allow reasonable redundancy based
on your specific product requirements when configuring the flash erasing timeout threshold, thus improving the reliability of your product.

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 needs to disable interrupts or scheduler before performing the flash operation.
- In a dual-core setup, the SDK needs to make sure that both CPUs are 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.

Related Documents

- Optional Features for Flash
- Concurrency Constraints for Flash on SPI

SPI Flash API ESP-IDF Version vs Chip-ROM Version  
There is a set of SPI Flash drivers in Chip-ROM which you can use by enabling `CONFIG_SPI_FLASH_ROM_IMPL`. Most of the ESP-IDF SPI Flash driver code are in internal RAM, therefore enabling this option frees some internal RAM usage. Note if you enable this option, this means some SPI Flash driver features and bugfixes that are done in ESP-IDF might not be included in the Chip-ROM version.

Feature Supported by ESP-IDF but Not in Chip-ROM

- Octal Flash chip support. See [OPI flash Support](#) for details.
- 32-bit-address support for GD25Q256. Note this feature is an optional feature, please do read [32-bit Address Flash Chips](#) for details.
- TH Flash chip support.
- Kconfig option `CONFIG_SPI_FLASH_CHECK_ERASE_TIMEOUT_DISABLED`.
- `CONFIG_SPI_FLASH_VERIFY_WRITE`, enabling this option helps you detect bad writing.
- `CONFIG_SPI_FLASH_LOG_FAILED_WRITE`, enabling this option prints the bad writing.
- `CONFIG_SPI_FLASH_WARN_SETTING_ZERO_TO_ONE`, enabling this option checks if you are writing zero to one.
- `CONFIG_SPI_FLASH_DANGEROUS_WRITE`, enabling this option checks for flash programming to certain protected regions like bootloader, partition table or application itself.
- `CONFIG_SPI_FLASH_ENABLE_COUNTERS`, enabling this option to collect performance data for ESP-IDF SPI Flash driver APIs.
• **CONFIG_SPI_FLASH_AUTO_SUSPEND**, enabling this option to automatically suspend / resume a long Flash operation when short Flash operation happens. Note this feature is an optional feature, please do read Auto Suspend & Resume for more limitations.

**Bugfixes Introduced in ESP-IDF but Not in Chip-ROM**

• Detected Flash physical size correctly, for larger than 256MBit Flash chips. (Commit ID: b4964279d44f73cce7cfdf5cf684567fbd6f6d9e)
• Fixed issue that only 4MB virtual address ranges can be mapped to read-only data on Flash.

**API Reference - SPI Flash**

**Header File**

• components/spi_flash/include/esp_flash_spi_init.h
• This header file can be included with:

```c
#include "esp_flash_spi_init.h"
```
• This header file is a part of the API provided by the `spi_flash` component. To declare that your component depends on `spi_flash`, add the following to your CMakeLists.txt:

```cmake
REQUIRES spi_flash
```

or

```cmake
PRIV_REQUIRES spi_flash
```

**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`.

**Parameters**

• `out_chip` - Pointer to hold the initialized chip.
• `config` - Configuration of the chips to initialize.

**Returns**

• 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.

`esp_err_t spi_bus_remove_flash_device (esp_flash_t *chip)`

Remove a SPI Flash device from the SPI bus.

**Parameters**

• `chip` - The flash device to remove.

**Returns**

• ESP_ERR_INVALID_ARG: The chip is invalid.
• ESP_OK: success.

**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_io_num

GPIO pin to output the CS signal.

esp_flash_io_mode_t io_mode

IO mode to read from the Flash.

enum esp_flash_speed_s speed

Speed of the Flash clock. Replaced by freq_mhz.

int input_delay_ns

Input delay of the data pins, in ns. Set to 0 if unknown.

int cs_id

CS line ID, ignored when not host_id is not SPI1_HOST, or CONFIG_SPI_FLASH_SHARE_SPI1_BUS is enabled. In this case, the CS line used is automatically assigned by the SPI bus lock.

int freq_mhz

The frequency of flash chip(MHZ)

Header File

- components/spi_flash/include/esp_flash.h
- This header file can be included with:

```c
#include "esp_flash.h"
```

- This header file is a part of the API provided by the spi_flash component. To declare that your component depends on spi_flash, add the following to your CMakeLists.txt:

```cmake
REQUIRES spi_flash
```

or

```cmake
PRIV_REQUIRES spi_flash
```

Functions

```c
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.
Parameters `chip` – Pointer to SPI flash chip to use. If NULL, `esp_flash_default_chip` is substituted.

**Returns** `ESP_OK` on success, or a flash error code if initialisation fails.

bool `esp_flash_chip_driver_initialized` (const `esp_flash_t` *chip)
Check if appropriate chip driver is set.

**Parameters** `chip` – Pointer to SPI flash chip to use. If NULL, `esp_flash_default_chip` is substituted.

**Returns** true if set, otherwise false.

`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_id` – [out] Pointer to receive ID value.

**Returns** `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:** 1. 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.

a. The `out_size` returned only stands for the size in the binary image header. If you want to get the real size of the chip, please call `esp_flash_get_physical_size` instead.

**Parameters**
- `chip` – Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`
- `out_size` – [out] Detected size in bytes, standing for the size in the binary image header.

**Returns** `ESP_OK` on success, or a flash error code if operation failed.

`esp_err_t esp_flash_get_physical_size` (`esp_flash_t` *chip, uint32_t *flash_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.

**Parameters**
- `chip` – Pointer to identify flash chip. Must have been successfully initialised via `esp_flash_init()`

**Returns** `ESP_OK` on success, or a flash error code if operation failed.

`esp_err_t esp_flash_read_unique_chip_id` (`esp_flash_t` *chip, uint64_t *out_id)
Read flash unique ID via the common “RDUID” SPI flash command.

ID is a 64-bit value.
Note: This is an optional feature, which is not supported on all flash chips. READ PROGRAMMING GUIDE FIRST!

**Parameters**
- `chip` - Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init().
- `out_id` - [out] Pointer to receive unique ID value.

**Returns**
- ESP_OK on success, or a flash error code if operation failed.
- ESP_ERR_NOT_SUPPORTED if the chip doesn’t support read id.

```c
esp_err_t esp_flash_erase_chip(esp_flash_t *chip)
```
Erase flash chip contents.

**Parameters**
- `chip` - Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()

**Returns**
- ESP_OK on success,
- ESP_ERR_NOT_SUPPORTED if the chip is not able to perform the operation. This is indicated by WREN = 1 after the command is sent.
- ESP_ERR_NOT_ALLOWED if a read-only partition is present.
- Other flash error code if operation failed.

```c
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.

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.

**Parameters**
- `chip` - Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. 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.

**Returns**
- ESP_OK on success,
- ESP_ERR_NOT_SUPPORTED if the chip is not able to perform the operation. This is indicated by WREN = 1 after the command is sent.
- ESP_ERR_NOT_ALLOWED if the address range (start & start + len) overlaps with a read-only partition address space
- Other flash error code if operation failed.

```c
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_driv in use (via the ‘chip-driv’ field).
**write_protected** [out] Pointer to boolean, set to the value of the write protect flag.

**Returns** ESP_OK on success, or a flash error code if operation failed.

```c
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. If NULL, esp_flash_default_chip is substituted.
  Must have been successfully initialised via esp_flash_init()
- write_protect - Boolean value for the write protect flag

**Returns** ESP_OK on success, or a flash error code if operation failed.

```c
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).

**Parameters**
- chip - Pointer to identify flash chip. Must have been successfully initialised via esp_flash_init()
- out_regions - [out] Pointer to receive a pointer to the array of protectable regions of the chip.
- out_num_regions - [out] Pointer to an integer receiving the count of protectable regions in the array returned in `out_regions`.

**Returns** ESP_OK on success, or a flash error code if operation failed.

```c
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).

**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(...).
Chapter 2. API Reference

- **out_protected** – [out] Pointer to a flag which is set based on the protected status for this region.

**Returns** ESP_OK on success, or a flash error code if operation failed.

```c
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).

**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.

**Returns** ESP_OK on success, or a flash error code if operation failed.

```c
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.

**Note:** If on-chip flash encryption is used, this function returns raw (ie encrypted) data. Use the flash cache to transparently decrypt data.

**Parameters**
- **chip** – Pointer to identify flash chip. If NULL, esp_flash_default_chip is substituted. 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.

**Returns**
- 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.

```c
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. If NULL, esp_flash_default_chip is substituted. 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.

**Returns**
- ESP_OK on success
- ESP_FAIL, bad write, this will be detected only when CONFIG_SPI_FLASH_VERIFY_WRITE is enabled
- ESP_ERR_NOT_SUPPORTED if the chip is not able to perform the operation. This is indicated by WREN = 1 after the command is sent.
- ESP_ERR_NOT_ALLOWED if the address range (address – address + length) overlaps with a read-only partition address space
- Other flash error code if operation failed.

```c
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

**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.

**Returns**
- ESP_OK: on success
- ESP_FAIL: bad write, this will be detected only when CONFIG_SPI_FLASH_VERIFY_WRITE is enabled
- ESP_ERR_NOT_SUPPORTED: encrypted write not supported for this chip.
- ESP_ERR_INVALID_ARG: Either the address, buffer or length is invalid.
- ESP_ERR_NOT_ALLOWED if the address range (address – address + length) overlaps with a read-only partition address space

```c
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.

**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.

**Returns**
- ESP_OK: on success
- ESP_ERR_NOT_SUPPORTED: encrypted read not supported for this chip.

```c
static inline 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.

**Parameters** **chip** – Pointer to SPI flash chip to use. If NULL, esp.flash_default_chip is substituted.

**Returns** true if flash works in quad mode, otherwise false
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.

It’s in the public header because some instances should be allocated statically in the startup code. May be updated according to hardware version and new flash chip feature requirements, shouldn’t be treated as public API.

For advanced developers, you may replace some of them with your implementations at your own risk.

**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, uint32_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, uint32_t us)**

Delay for at least ‘us’ microseconds. Called in between ‘start’ and ‘end’.

**void *(*get_temp_buffer**(void *arg, size_t request_size, size_t *out_size)**

Called for get temp buffer when buffer from application cannot be directly read into/write from.

**void *(*release_temp_buffer**(void *arg, void *temp_buf)**

Called for release temp buffer.

**esp_err_t (**check_yield**(void *arg, uint32_t chip_status, uint32_t *out_request)**

Yield to other tasks. Called during erase operations.

Return ESP_OK means yield needs to be called (got an event to handle), while ESP_ERR_TIMEOUT means skip yield.

**esp_err_t (**yield**(void *arg, uint32_t *out_status)**

Yield to other tasks. Called during erase operations.
int64_t (*get_system_time)(void *arg)
Called for get system time.

void (*set_flash_op_status)(uint32_t op_status)
Call to set flash operation status

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()). It’s in the public header because some instances should be allocated statically in the startup code. May be updated according to hardware version and new flash chip feature requirements, shouldn’t be treated as public API.
For advanced developers, you may replace some of them with your implementations at your own risk.

Public Members

spi_flash_host_inst_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. Note: this stands for the size in the binary image header. If you want to get the flash physical size, please call esp_flash_get_physical_size.

uint32_t chip_id
Detected chip id.

uint32_t busy
This flag is used to verify chip’s status.

uint32_t hpm_dummy_ena
This flag is used to verify whether flash works under HPM status.

uint32_t reserved_flags
reserved.
Chapter 2. API Reference

Macros

SPI_FLASH_YIELD_REQ_YIELD

SPI_FLASH_YIELD_REQ_SUSPEND

SPI_FLASH_YIELD_STA_RESUME

SPI_FLASH_OS_IS_ERASING_STATUS_FLAG

Type Definitions

typedef struct spi_flash_chip_t spi_flash_chip_t

Header File

- components/spi_flash/include/spi_flash_mmap.h
- This header file can be included with:

```c
#include "spi_flash_mmap.h"
```
- This header file is a part of the API provided by the spi_flash component. To declare that your component depends on spi_flash, add the following to your CMakeLists.txt:

```cmake
REQUIRES spi_flash
```
- or

```cmake
PRIV_REQUIRES spi_flash
```

Functions

`esp_err_t spi_flash_mmap(size_t src_addr, size_t size, spi_flash_mmap_memory_t memory, const void **out_ptr, spi_flash_mmap_handle_t *out_handle)`

Map region of flash memory into data or instruction address space.

This function allocates sufficient number of 64kB MMU pages and configures them to map the requested region of flash memory into the address space. It may reuse MMU pages which already provide the required mapping.

As with any allocator, if mmap/munmap are heavily used then the address space may become fragmented. To troubleshoot issues with page allocation, use `spi_flash_mmap_dump()` function.

**Parameters**

- **src_addr** - Physical address in flash where requested region starts. This address must be aligned to 64kB boundary (SPI_FLASH_MMU_PAGE_SIZE)
- **size** - Size of region to be mapped. This size will be rounded up to a 64kB boundary
- **memory** - Address space where the region should be mapped (data or instruction)
- **out_ptr** - [out] Output, pointer to the mapped memory region
- **out_handle** - [out] Output, handle which should be used for `spi_flash_mmap` call

**Returns**

ESP_OK on success, ESP_ERR_NO_MEM if pages cannot be allocated

`esp_err_t spi_flash_mmap_pages(const int *pages, size_t page_count, spi_flash_mmap_memory_t memory, const void **out_ptr, spi_flash_mmap_handle_t *out_handle)`

Map sequences of pages of flash memory into data or instruction address space.

This function allocates sufficient number of 64kB MMU pages and configures them to map the indicated pages of flash memory contiguously into address space. In this respect, it works in a similar way as `spi_flash_mmap()` but it allows mapping a (maybe non-contiguous) set of pages into a contiguous region of memory.

**Parameters**
• **pages** – An array of numbers indicating the 64kB pages in flash to be mapped contiguously into memory. These indicate the indexes of the 64kB pages, not the byte-size addresses as used in other functions. Array must be located in internal memory.

• **page_count** – Number of entries in the pages array

• **memory** – Address space where the region should be mapped (instruction or data)

• **out_ptr** – [out] Output, pointer to the mapped memory region

• **out_handle** – [out] Output, handle which should be used for `spi_flash_munmap` call

**Returns**

• ESP_OK on success

• ESP_ERR_NO_MEM if pages cannot be allocated

• ESP_ERR_INVALID_ARG if pagecount is zero or pages array is not in internal memory

```c
void spi_flash_munmap (spi_flash_mmap_handle_t handle)
```

Release region previously obtained using `spi_flash_mmap`.

**Note:** Calling this function will not necessarily unmmap memory region. Region will only be unmapped when there are no other handles which reference this region. In case of partially overlapping regions it is possible that memory will be unmapped partially.

**Parameters handle** – Handle obtained from `spi_flash_mmap`

```c
void spi_flash_mmap_dump (void)
```

Display information about mapped regions.

This function lists handles obtained using `spi_flash_mmap`, along with range of pages allocated to each handle. It also lists all non-zero entries of MMU table and corresponding reference counts.

```c
uint32_t spi_flash_mmap_get_free_pages (spi_flash_mmap_memory_t memory)
```

get free pages number which can be mmap

This function will return number of free pages available in mmu table. This could be useful before calling actual `spi_flash_mmap` (maps flash range to DCache or ICache memory) to check if there is sufficient space available for mapping.

**Parameters memory** – memory type of MMU table free page

**Returns** number of free pages which can be mmaped

```c
size_t spi_flash_cache2phys (const void* cached)
```

Given a memory address where flash is mapped, return the corresponding physical flash offset.

Cache address does not have have been assigned via `spi_flash_mmap()`, any address in memory mapped flash space can be looked up.

**Parameters cached** – Pointer to flashed cached memory.

**Returns**

• SPI_FLASH_CACHE2PHYS_FAIL If cache address is outside flash cache region, or the address is not mapped.

• Otherwise, returns physical offset in flash

```c
const void *spi_flash_phys2cache (size_t phys_offs, spi_flash_mmap_memory_t memory)
```

Given a physical offset in flash, return the address where it is mapped in the memory space.

Physical address does not have to have been assigned via `spi_flash_mmap()`; any address in flash can be looked up.

**Note:** Only the first matching cache address is returned. If MMU flash cache table is configured so multiple entries point to the same physical address, there may be more than one cache address corresponding to that physical address. It is also possible for a single physical address to be mapped to both the IROM and DROM regions.
Note: This function doesn’t impose any alignment constraints, but if memory argument is SPI_FLASH_MMAP_INST and phys_offs is not 4-byte aligned, then reading from the returned pointer will result in a crash.

Parameters
- **phys_offs** – Physical offset in flash memory to look up.
- **memory** – Address space type to look up a flash cache address mapping for (instruction or data)

Returns
- NULL if the physical address is invalid or not mapped to flash cache of the specified memory type.
- Cached memory address (in IROM or DROM space) corresponding to phys_offs.

Macros

**ESP_ERR_FLASH_OP_FAIL**
This file contains spi_flash_mmap_xx APIs, mainly for doing memory mapping to an SPI0-connected external Flash, as well as some helper functions to convert between virtual and physical address.

ESP_ERR_FLASH_OP_TIMEOUT

SPI_FLASH_SEC_SIZE
SPI Flash sector size

SPI_FLASH_MMU_PAGE_SIZE
Flash cache MMU mapping page size

SPI_FLASH_CACHE2PHYS_FAIL

Type Definitions

typedef uint32_t spi_flash_mmap_handle_t
Opaque handle for memory region obtained from spi_flash_mmap.

Enumerations

enum spi_flash_mmap_memory_t
Enumeration which specifies memory space requested in an mmap call.

Values:

enumerator SPI_FLASH_MMAP_DATA
map to data memory, allows byte-aligned access

enumerator SPI_FLASH_MMAP_INST
map to instruction memory, allows only 4-byte-aligned access

Header File

- components/hal/include/hal/spi_flash_types.h
- This header file can be included with:
Chapter 2. API Reference

```
#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 reserved`
  - Reserved, must be 0.

- `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

- `uint32_t flags`
  - Flags for this transaction. Set to 0 for now.

- `uint16_t command`
  - Command to send.

- `uint8_t dummy_bitlen`
  - Basic dummy bits to use.

- `uint32_t io_mode`
  - Flash working mode when SPI_FLASH_IGNORE_BASEIO is specified.

struct **spi_flash_sus_cmd_conf**

Configuration structure for the flash chip suspend feature.

**Public Members**
uint32_t sus_mask
    SUS/SUS1/SUS2 bit in flash register.

uint32_t cmd_rdsr
    Read flash status register(2) command.

uint32_t sus_cmd
    Flash suspend command.

uint32_t res_cmd
    Flash resume command.

uint32_t reserved
    Reserved, set to 0.

struct spi_flash_encryption_t
    Structure for flash encryption operations.

Public Members

void (*flash_encryption_enable)(void)
    Enable the flash encryption.

void (*flash_encryption_disable)(void)
    Disable the flash encryption.

void (*flash_encryption_data_prepare)(uint32_t address, const uint32_t *buffer, uint32_t size)
    Prepare flash encryption before operation.

Note: address and buffer must be 8-word aligned.

    Param address  The destination address in flash for the write operation.
    Param buffer   Data for programming
    Param size     Size to program.

void (*flash_encryption_done)(void)
    Flash data encryption operation is done.

void (*flash_encryption_destroy)(void)
    Destroy encrypted result

bool (*flash_encryption_check)(uint32_t address, uint32_t length)
    Check if is qualified to encrypt the buffer

    Param address  the address of written flash partition.
    Param length   Buffer size.

struct spi_flash_host_inst_t
    SPI Flash Host driver instance
Public Members

const struct spi_flash_host_driver_s *driver
Pointer to the implementation function table.

struct spi_flash_host_driver_s
Host driver configuration and context structure.

Public Members

esp_err_t (*dev_config)(spi_flash_host_inst_t *host)
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_inst_t *host, spi_flash_trans_t *t)
Send an user-defined spi transaction to the device.

esp_err_t (*read_id)(spi_flash_host_inst_t *host, uint32_t *id)
Read flash ID.

void (*erase_chip)(spi_flash_host_inst_t *host)
Erase whole flash chip.

void (*erase_sector)(spi_flash_host_inst_t *host, uint32_t start_address)
Erase a specific sector by its start address.

void (*erase_block)(spi_flash_host_inst_t *host, uint32_t start_address)
Erase a specific block by its start address.

esp_err_t (*read_status)(spi_flash_host_inst_t *host, uint8_t *out_sr)
Read the status of the flash chip.

esp_err_t (*set_write_protect)(spi_flash_host_inst_t *host, bool wp)
Disable write protection.

void (*program_page)(spi_flash_host_inst_t *host, 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_inst_t *host, const void *p)
Check whether the SPI host supports direct write.
When cache is disabled, SPI1 doesn’t support directly write when buffer isn’t internal.

int (*write_data_slicer)(spi_flash_host_inst_t *host, uint32_t address, uint32_t len, uint32_t *align_addr, uint32_t page_size)
Slicer for write data. The program_page should be called iteratively with the return value of this function.

Param address Beginning flash address to write
Param len Length request to write
Param align_addr Output of the aligned address to write to
**Param page_size**  Physical page size of the flash chip  
**Return**  Length that can be actually written in one program_page call

```c
esp_err_t (*read)(spi_flash_host_inst_t *host, void *buffer, uint32_t address, uint32_t read_len)  
Read data from the flash. Check max_read_bytes for the maximum allowed reading length.
```

```c
bool (*supports_direct_read)(spi_flash_host_inst_t *host, const void *p)  
Check whether the SPI host supports direct read.  
When cache is disabled, SPI1 doesn’t support directly read when the given buffer isn’t internal.
```

```c
int (*read_data_slicer)(spi_flash_host_inst_t *host, uint32_t address, uint32_t len, uint32_t *align_addr, uint32_t page_size)  
Slicer for read data. The read should be called iteratively with the return value of this function.
```

```c
Param address  Beginning flash address to read  
Param len  Length request to read  
Param align_addr  Output of the aligned address to read  
Param page_size  Physical page size of the flash chip  
Return  Length that can be actually read in one read call
```

```c
uint32_t (*host_status)(spi_flash_host_inst_t *host)  
Check the host status, 0:busy, 1:idle, 2:suspended.
```

```c
esp_err_t (*configure_host_io_mode)(spi_flash_host_inst_t *host, 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.
```

```c
void (*poll_cmd_done)(spi_flash_host_inst_t *host)  
Internal use, poll the HW until the last operation is done.
```

```c
esp_err_t (*flush_cache)(spi_flash_host_inst_t *host, 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.
```

```c
void (*check_suspend)(spi_flash_host_inst_t *host)  
Suspend check erase/program operation, reserved for ESP32-C3 and ESP32-S3 spi flash ROM IMPL.
```

```c
void (*resume)(spi_flash_host_inst_t *host)  
Resume flash from suspend manually
```

```c
void (*suspend)(spi_flash_host_inst_t *host)  
Set flash in suspend status manually
```

```c
esp_err_t (*sus_setup)(spi_flash_host_inst_t *host, const spi_flash_sus_cmd_conf *sus_conf)  
Suspend feature setup for setting cmd and status register mask.
```

**Macros**

**SPI_FLASH_TRANS_FLAG_CMD16**  
Send command of 16 bits.
SPI_FLASH_TRANS_FLAG_IGNORE_BASEIO
Not applying the basic io mode configuration for this transaction.

SPI_FLASH_TRANS_FLAG_BYTE_SWAP
Used for DTR mode, to swap the bytes of a pair of rising/falling edge.

SPI_FLASH_TRANS_FLAG_PE_CMD
Indicates that this transaction is to erase/program flash chip.

SPI_FLASH_CONFIG_CONF_BITS
OR the io_mode with this mask, to enable the dummy output feature or replace the first several dummy bits into address to meet the requirements of conf bits. (Used in DIO/QIO/OIO mode)

SPI_FLASH_OPI_FLAG
A flag for flash work in opi mode, the io_mode below are opi, above are SPI/QSPI mode. DO NOT use this value in any API.

SPI_FLASH_READ_MODE_MIN
Slowest io_mode supported by ESP32, currently SlowRd.

Type Definitions
typedef enum esp_flash_speed_s 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_FLSH_SPEED_MAX-1 or highest frequency supported by your flash, and decrease the speed until the probing success.

typedef struct spi_flash_host_driver_s spi_flash_host_driver_t

Enumerations
enum esp_flash_speed_s
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_FLSH_SPEED_MAX-1 or highest frequency supported by your flash, and decrease the speed until the probing success.

Values:

counter ESP_FLASH_5MHZ
The flash runs under 5MHz.

counter ESP_FLASH_10MHZ
The flash runs under 10MHz.

counter ESP_FLASH_20MHZ
The flash runs under 20MHz.
enumerator **ESP_FLASH_26MHZ**
The flash runs under 26MHz.

enumerator **ESP_FLASH_40MHZ**
The flash runs under 40MHz.

enumerator **ESP_FLASH_80MHZ**
The flash runs under 80MHz.

enumerator **ESP_FLASH_120MHZ**
The flash runs under 120MHz. 120MHZ can only be used by main flash after timing tuning in system. Do not use this directly in any API.

enumerator **ESP_FLASH_SPEED_MAX**
The maximum frequency supported by the host is **ESP_FLASH_SPEED_MAX-1**.

```c
enum esp_flash_io_mode_t
```

Mode used for reading from SPI flash.

**Values:**

enumerator **SPI_FLASH_SLOWRD**
Data read using single I/O, some limits on speed.

enumerator **SPI_FLASH_FASTRD**
Data read using single I/O, no limit on speed.

enumerator **SPI_FLASH_DOUT**
Data read using dual I/O.

enumerator **SPI_FLASH_DIO**
Both address & data transferred using dual I/O.

enumerator **SPI_FLASH_QOUT**
Data read using quad I/O.

enumerator **SPI_FLASH_QIO**
Both address & data transferred using quad I/O.

enumerator **SPI_FLASH_OPI_STR**
Only support on OPI flash, flash read and write under STR mode.

enumerator **SPI_FLASH_OPI_DTR**
Only support on OPI flash, flash read and write under DTR mode.

enumerator **SPI_FLASH_READ_MODE_MAX**
The fastest io mode supported by the host is **ESP_FLASH_READ_MODE_MAX-1**.
Chapter 2. API Reference

Header File

- components/hal/include/hal/esp_flash_err.h
- This header file can be included with:

```c
#include "hal/esp_flash_err.h"
```

Macros

**ESP_ERR_FLASH_NOT_INITIALIZED**

- esp_flash_chip_t structure not correctly initialised by esp_flash_init().

**ESP_ERR_FLASH_UNSUPPORTED_HOST**

- Requested operation isn’t supported via this host SPI bus (chip->spi field).

**ESP_ERR_FLASH_UNSUPPORTED_CHIP**

- Requested operation isn’t supported by this model of SPI flash chip.

**ESP_ERR_FLASH_PROTECTED**

- Write operation failed due to chip’s write protection being enabled.

Enumerations

```c
eenum [anonymous]

Values:

enumerator ESP_ERR_FLASH_SIZE_NOT_MATCH
- The chip doesn’t have enough space for the current partition table.

enumerator ESP_ERR_FLASH_NO_RESPONSE
- Chip did not respond to the command, or timed out.
```

Header File

- components/spi_flash/include/esp_spi_flash_counters.h
- This header file can be included with:

```c
#include "esp_spi_flash_counters.h"
```

- This header file is a part of the API provided by the spi_flash component. To declare that your component depends on spi_flash, add the following to your CMakeLists.txt:

```cmake
REQUIRES spi_flash
```

or

```cmake
PRIV_REQUIRES spi_flash
```

Functions

```c
void esp_flash_reset_counters (void)
- Reset SPI flash operation counters.

void spi_flash_reset_counters (void)

void esp_flash_dump_counters (FILE *stream)
- Print SPI flash operation counters.
```
void spi_flash_dump_counters (void)

const esp_flash_counters_t *esp_flash_get_counters (void)

    Returns pointer to the esp_flash_counters_t structure holding values of the operation counters

const spi_flash_counters_t *spi_flash_get_counters (void)

Structures

struct esp_flash_counter_t

    Structure holding statistics for one type of operation

Public Members

    uint32_t count
            number of times operation was executed

    uint32_t time
            total time taken, in microseconds

    uint32_t bytes
            total number of bytes

struct esp_flash_counters_t

    Structure for counters of flash actions

Public Members

    esp_flash_counter_t read
            counters for read action, like esp_flash_read

    esp_flash_counter_t write
            counters for write action, like esp_flash_write

    esp_flash_counter_t erase
            counters for erase action, like esp_flash_erase

Type Definitions

typedef esp_flash_counter_t spi_flash_counter_t

typedef esp_flash_counters_t spi_flash_counters_t

API Reference - Flash Encrypt

Header File

    • components/bootloader_support/include/esp_flash_encrypt.h
    • This header file can be included with:
This header file is a part of the API provided by the bootloader_support component. To declare that your component depends on bootloader_support, add the following to your CMakeLists.txt:

```cmake
REQUIRES bootloader_support
```

or

```cmake
PRIV_REQUIRES bootloader_support
```

### Functions

**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.

- **Returns**: true if flash encryption is enabled.

**esp_err_t esp_flash_encrypt_check_and_update (void)**

- **Returns**: true - Flash Encryption is enabled False - Flash Encryption is not enabled

**bool esp_flash_encrypt_state (void)**

Returns the Flash Encryption state and prints it.

- **Returns**: True - Flash Encryption is enabled False - Flash Encryption is not enabled

**bool esp_flash_encrypt_initialized_once (void)**

Checks if the first initialization was done.

If the first initialization was done then FLASH_CRYPT_CNT != 0

- **Returns**: true - the first initialization was done false - the first initialization was NOT done

**esp_err_t esp_flash_encrypt_init (void)**

The first initialization of Flash Encryption key and related eFuses.

- **Returns**: ESP_OK if all operations succeeded

**esp_err_t esp_flash_encrypt_contents (void)**

Encrypts flash content.

- **Returns**: ESP_OK if all operations succeeded

**esp_err_t esp_flash_encrypt_enable (void)**

Activates Flash encryption on the chip.

It burns FLASH_CRYPT_CNT eFuse based on the CONFIG_SECURE_FLASH_ENCRYPTION_MODE_RELEASE option.

- **Returns**: ESP_OK if all operations succeeded

**bool esp_flash_encrypt_is_write_protected (bool print_error)**

Returns True if the write protection of FLASH_CRYPT_CNT is set.

- **Parameters**: print_error - Print error if it is write protected

- **Returns**: true - if FLASH_CRYPT_CNT is write protected

**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.

- **Parameters**
  - **src_addr** - Source offset in flash. Should be multiple of 4096 bytes.

---

Submit Document Feedback
Chapter 2. API Reference

- **data_length** – Length of data to encrypt in bytes. Will be rounded up to next multiple of 4096 bytes.

  **Returns** ESP_OK if all operations succeeded, ESP_ERR_FLASH_OP_FAIL if SPI flash fails, ESP_ERR_FLASH_OP_TIMEOUT if flash times out.

```c
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.

```c
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.

**Returns**

```c
void esp_flash_encryption_init_checks (void)
```

Check the flash encryption mode during startup.

Verifies the flash encryption config during startup:

- 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

**Note:** This function is called automatically during app startup, it doesn’t need to be called from the app.

```c
esp_err_t esp_flash_encryption_enable_secure_features (void)
```

Set all secure eFuse features related to flash encryption.

**Returns**

- ESP_OK - Successfully

```c
bool esp_flash_encryption_cfg_verify_release_mode (void)
```

Returns the verification status for all physical security features of flash encryption in release mode.

If the device has flash encryption feature configured in the release mode, then it is highly recommended to call this API in the application startup code. This API verifies the sanity of the eFuse configuration against the release (production) mode of the flash encryption feature.

**Returns**

- True - all eFuses are configured correctly
- False - not all eFuses are configured correctly.

```c
void esp_flash_encryption_set_release_mode (void)
```

Switches Flash Encryption from “Development” to “Release”.

If already in “Release” mode, the function will do nothing. If flash encryption efuse is not enabled yet then abort. It burns:

- ” disable encrypt in dl mode”
- set FLASH_CRYPT_CNT efuse to max
Enumerations

```c
enum esp_flash_enc_mode_t
{
  # Values:
  ESP_FLASH_ENC_MODE_DISABLED,
  ESP_FLASH_ENC_MODE_DEVELOPMENT,
  ESP_FLASH_ENC_MODE_RELEASE,
};
```

2.5.20 SPI Master Driver

SPI Master driver is a program that controls ESP32-S2’s General Purpose SPI (GP-SPI) peripheral(s) when it functions as a master.

For more hardware information about the GP-SPI peripheral(s), see ESP32-S2 Technical Reference Manual > SPI Controller [PDF].

Terminology

The terms used in relation to the SPI Master driver are given in the table below.
Table: API Reference

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>The SPI controller peripheral inside ESP32-S2 initiates SPI transmissions over the bus and acts as an SPI Master.</td>
</tr>
<tr>
<td>Device</td>
<td>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.</td>
</tr>
<tr>
<td>Bus</td>
<td>A signal bus, common to all Devices connected to one Host. In general, a bus includes the following lines: MOSI, 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 a daisy-chain manner.</td>
</tr>
<tr>
<td>MOSI</td>
<td>Master Out, Slave In, a.k.a. D. Data transmission from a Host to Device. Also data0 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>MISO</td>
<td>Master In, Slave Out, a.k.a. Q. Data transmission from a Device to Host. Also data1 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>SCLK</td>
<td>Serial Clock. The oscillating signal generated by a Host keeps the transmission of data bits in sync.</td>
</tr>
<tr>
<td>CS</td>
<td>Chip Select. Allows a Host to select individual Device(s) connected to the bus in order to send or receive data.</td>
</tr>
<tr>
<td>QUADWP</td>
<td>Write Protect signal. Used for 4-bit (qio/qout) transactions. Also for the data2 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>QUADHD</td>
<td>Hold signal. Used for 4-bit (qio/qout) transactions. Also for the data3 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>DATA4</td>
<td>Data4 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>DATA5</td>
<td>Data5 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>DATA6</td>
<td>Data6 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>DATA7</td>
<td>Data7 signal in Octal/OPI mode.</td>
</tr>
<tr>
<td>Assertion</td>
<td>The action of activating a line.</td>
</tr>
<tr>
<td>De-assertion</td>
<td>The action of returning the line back to inactive (back to idle) status.</td>
</tr>
<tr>
<td>Transaction</td>
<td>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.</td>
</tr>
<tr>
<td>Launch Edge</td>
<td>Edge of the clock at which the source register launches the signal onto the line.</td>
</tr>
<tr>
<td>Latch Edge</td>
<td>Edge of the clock at which the destination register latches in the signal.</td>
</tr>
</tbody>
</table>

**Driver Features**

The SPI Master driver governs the communications between Hosts and 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, see *SPI Bus Lock*.

**Warning:** The SPI Master driver allows multiple Devices to be connected on a same SPI bus (sharing a single ESP32-S2 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. You can use `spi_bus_config_t::isr_cpu_id` to register the SPI ISR to the same core as SPI peripheral-related tasks to ensure thread safety.
- Add a mutex lock around the shared Device using `xSemaphoreCreateMutex`. 

---

*Espressif Systems* 712  Release master

Submit Document Feedback
SPI Features

SPI Master

SPI Bus Lock  To realize the multiplexing of different devices from different drivers, including SPI Master, SPI Flash, etc., an SPI bus lock is applied on each SPI bus. Drivers can attach their devices to the bus with the arbitration of the lock.

Each bus lock is initialized with a BG (background) service registered. All devices that request transactions on the bus should wait until the BG is successfully disabled.

- For the SPI1 bus, the BG is the cache. The bus lock disables the cache before device operations start, and enables it again after the device releases the lock. No devices on SPI1 are allowed to use ISR, since it is meaningless for the task to yield to other tasks when the cache is disabled. The SPI Master driver has not supported SPI1 bus. Only the SPI Flash driver can attach to the bus.
- For other buses, the driver can register the ISR as a BG. If a device task requests exclusive bus access, the bus lock will block the task, disable the ISR, and then unblock the task. After the task releases the lock, the lock will try to re-enable the ISR if there are still pending transactions in the ISR.

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.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>In this phase, a command (0-16 bit) is written to the bus by the Host.</td>
</tr>
<tr>
<td>Address</td>
<td>In this phase, an address (0-32 bit) is transmitted over the bus by the Host.</td>
</tr>
<tr>
<td>Dummy</td>
<td>This phase is configurable and is used to meet the timing requirements.</td>
</tr>
<tr>
<td>Write</td>
<td>Host sends data to a Device. This data follows the optional command and address phases and is indistinguishable from them at the electrical level.</td>
</tr>
<tr>
<td>Read</td>
<td>Device sends data to its Host.</td>
</tr>
</tbody>
</table>

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 `spi_transaction_t::length` and `spi_transaction_t::rxlength` 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 `spi_device_interface_config_t::command_bits` and/or `spi_device_interface_config_t::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 `spi_transaction_t::rx_buffer` is NULL and `SPI_TRANS_USE_RXDATA` is not set, the Read phase is skipped. If `spi_transaction_t::tx_buffer` is NULL and `SPI_TRANS_USE_TXDATA` is not set, the Write phase is skipped.
The driver supports two types of transactions: 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 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 automatically handles them one by one in the interrupt service routine (ISR). It allows the task to switch to other procedures until all the transactions are 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 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 duration. 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 µs 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.

**Transaction Line Mode** Supported line modes for ESP32-S2 are listed as follows, to make use of these modes, set the member `flags` in the struct `spi_transaction_t` as shown in the Transaction Flag column. If you want to check if corresponding IO pins are set or not, set the member `flags` in the `spi_bus_config_t` as shown in the Bus IO setting Flag column.

<table>
<thead>
<tr>
<th>Mode name</th>
<th>Command Line Width</th>
<th>Address Line Width</th>
<th>Data Line Width</th>
<th>Transaction Flag</th>
<th>Bus IO Setting Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal SPI</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dual Output</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>SPI_TRANS_MODE_DUAL</td>
<td>SPICOM-MON_BUSFLAG_DUAL</td>
</tr>
<tr>
<td>Dual I/O</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>SPI_TRANS_MODE_DUAL</td>
<td>SPICOM-MON_ADDRFLAG_DUAL</td>
</tr>
<tr>
<td>Quad Output</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>SPI_TRANS_MODE_DUAL</td>
<td>SPICOM-MON_BUSFLAG_QUAD</td>
</tr>
<tr>
<td>Quad I/O</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>SPI_TRANS_MODE_DUAL</td>
<td>SPICOM-MON_ADDRFLAG_QUAD</td>
</tr>
<tr>
<td>Octal Output</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>SPI_TRANS_MODE_DUAL</td>
<td>SPICOM-MON_BUSFLAG_OCTAL</td>
</tr>
<tr>
<td>OPI</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>SPI_TRANS_MODE_DUAL</td>
<td>SPICOM-MON_ADDRFLAG_OCTAL</td>
</tr>
</tbody>
</table>

**Command and Address Phases** During the Command and Address phases, the members `spi_transaction_t::cmd` and `spi_transaction_t::addr` 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...
Chapter 2. API Reference

of base as usual. Then the length of each phase will be equal to `spi_transaction_ext_t::command_bits` and `spi_transaction_ext_t::address_bits` set in the struct `spi_transaction_ext_t`.

If the Command and Address phase need to have the same number of lines as the data phase, you need to set `SPI_TRANS_MULTILINE_CMD` and/or `SPI_TRANS_MULTILINE_ADDR` to the `flags` member in the struct `spi_transaction_t`. Also see `Transaction Line Mode`.

**Write and Read Phases** Normally, the data that needs to be transferred to or from a Device is read from or written to a chunk of memory indicated by the members `spi_transaction_t::rx_buffer` and `spi_transaction_t::tx_buffer`. If DMA is enabled for transfers, the buffers are required to be:

1. Allocated in DMA-capable internal memory (MALLOC_CAP_DMA), see `DMA-Capable Memory`.
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.

If using more than one data line to transmit, please set `SPI_DEVICE_HALFDUPLEX` flag for the member `flags` in the struct `spi_device_interface_config_t`. And the member `flags` in the struct `spi_transaction_t` should be set as described in `Transaction Line Mode`.

**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 remove a certain Device from the bus, call `spi_bus_remove_device()` with the Device handle as an argument.
- (Optional) To remove the driver from the bus, make sure no more devices 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 `spi_transaction_t::tx_data` member and setting the `SPI_TRANS_USE_TXDATA` flag on the transmission. For received data, use `spi_transaction_t::rx_data` and set `SPI_TRANS_USE_RXDATA`. In both cases, do not touch the `spi_transaction_t::tx_buffer` or `spi_transaction_t::rx_buffer`.  

Espressif Systems 715 Release master  
Submit Document Feedback
members, because they use the same memory locations as \texttt{spi_transaction_t::tx_data} and \texttt{spi_transaction_t::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 is first used in rare cases. If a value of fewer than 8 bits needs to be sent, the bits should be written into memory in the MSB first manner.

For example, if \texttt{0b000010} needs to be sent, it should be written into a \texttt{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-S2 is a little-endian chip, which means that the least significant byte of \texttt{uint16_t} and \texttt{uint32_t} variables is stored at the smallest address. Hence, if \texttt{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 a size differing from \texttt{uint8_t} arrays, the following macros can be used to transform data to the format that can be sent by the SPI driver directly:

- \texttt{SPI_SWAP_DATA_TX} for data to be transmitted
- \texttt{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 transaction (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 \texttt{spi_device_polling_end()} after \texttt{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 \texttt{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 \texttt{spi_device_polling_start()}. To do that, you can keep calling \texttt{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 the chip’s peripheral signals have a 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.

When an SPI Host is set to 80 MHz or lower frequencies, routing SPI pins via the GPIO matrix will behave the same compared to routing them via IOMUX.

The IO_MUX pins for SPI buses are given below.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>GPIO Number (SPI2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS0(^1)</td>
<td>10</td>
</tr>
<tr>
<td>SCLK</td>
<td>12</td>
</tr>
<tr>
<td>MISO</td>
<td>13</td>
</tr>
<tr>
<td>MOSI</td>
<td>11</td>
</tr>
<tr>
<td>QUADWP</td>
<td>14</td>
</tr>
<tr>
<td>QUADHD</td>
<td>9</td>
</tr>
</tbody>
</table>

\(^1\) 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 Duration  Transaction duration includes setting up SPI peripheral registers, copying data to FIFOs or setting up DMA links, and the time for SPI transactions.

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 CPU time but increases the transaction duration. 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 µs 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 µs, but the transaction length is limited to 64 bytes for both write and read.

The typical transaction duration for one byte of data is given below.

- Interrupt Transaction via DMA: 23 µs.
- Interrupt Transaction via CPU: 22 µs.
- Polling Transaction via DMA: 9 µs.
- Polling Transaction via CPU: 8 µs.

Note that these data are tested with CONFIG_SPI_MASTER_ISR_IN_IRAM enabled. SPI transaction related code are placed in the internal memory. If this option is turned off (for example, for internal memory optimization), the transaction duration may be affected.

SPI Clock Frequency  The clock source of the GPSPI peripherals can be selected by setting spi_device_handle_t::cfg::clock_source. You can refer to spi_clock_source_t to know the supported clock sources.

By default driver sets spi_device_handle_t::cfg::clock_source to SPI_CLK_SRC_DEFAULT. This usually stands for the highest frequency among GP SPI clock sources. Its value is different among chips.

The actual clock frequency of a Device may not be exactly equal to the number you set, it is re-calculated by the driver to the nearest hardware-compatible number, and not larger than the clock frequency of the clock source. You can call spi_device_get_actual_freq() to know the actual frequency computed by the driver.

The theoretical maximum transfer speed of the Write or Read phase can be calculated according to the table below:

<table>
<thead>
<tr>
<th>Line Width of Write/Read phase</th>
<th>Speed (Bps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Line</td>
<td>SPI Frequency / 8</td>
</tr>
<tr>
<td>2-Line</td>
<td>SPI Frequency / 4</td>
</tr>
<tr>
<td>4-Line</td>
<td>SPI Frequency / 2</td>
</tr>
<tr>
<td>8-Line</td>
<td>SPI Frequency</td>
</tr>
</tbody>
</table>

The transfer speed calculation of other phases (Command, Address, Dummy) is similar.
Cache Missing  The default config puts only the ISR into the IRAM. Other SPI-related functions, including the driver itself and the callback, might suffer from cache misses and need to wait until the code is read from 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 missing.

Note:  SPI driver implementation is based on FreeRTOS APIs, to use `CONFIG_SPI_MASTER_IN_IRAM`, you should not enable `CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH`.

For an interrupt transaction, the overall cost is \(20+8n/F_{spi}\) [µs] 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.

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>Transaction Interval (µs)</th>
<th>Transaction Length (bytes)</th>
<th>Total Time (µs)</th>
<th>Total Speed (KBps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>1</td>
<td>26</td>
<td>38.5</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>8</td>
<td>33</td>
<td>242.4</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>16</td>
<td>41</td>
<td>490.2</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>64</td>
<td>89</td>
<td>719.1</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>128</td>
<td>153</td>
<td>836.6</td>
</tr>
</tbody>
</table>

When a transaction length is short, the cost of the 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 are 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 missing. For more details, see `IRAM-Safe Interrupt Handlers`.

Application Example

The code example for using the SPI master half duplex mode to read/write an AT93C46D EEPROM (8-bit mode) can be found in the `peripherals/spi_master/hd_eeprom` directory of ESP-IDF examples.

The code example for using the SPI master full duplex mode to drive a SPI_LCD (e.g. ST7789V or ILI9341) can be found in the `peripherals/spi_master/lcd` directory of ESP-IDF examples.

API Reference - SPI Common

Header File

- `components/hal/include/hal/spi_types.h`
- This header file can be included with:

```c
#include "hal/spi_types.h"
```

Structures

`struct spi_line_mode_t`

Line mode of SPI transaction phases: CMD, ADDR, DOUT/DIN.

Public Members
uint8_t cmd_lines  
    The line width of command phase, e.g. 2-line-cmd-phase.

uint8_t addr_lines  
    The line width of address phase, e.g. 1-line-addr-phase.

uint8_t data_lines  
    The line width of data phase, e.g. 4-line-data-phase.

**Type Definitions**

typedef `soc_periph_spi_clk_src_t` spi_clock_source_t  
    Type of SPI clock source.

**Enumerations**

enum spi_host_device_t  
    Enum with the three SPI peripherals that are software-accessible in it.

    Values:

    enumerator SPI1_HOST  
        SPI1.

    enumerator SPI2_HOST  
        SPI2.

    enumerator SPI3_HOST  
        SPI3.

    enumerator SPI_HOST_MAX  
        invalid host value

denum spi_event_t  
    SPI Events.

    Values:

    enumerator SPI_EV_BUF_TX  
        The buffer has sent data to master.

    enumerator SPI_EV_BUF_RX  
        The buffer has received data from master.

    enumerator SPI_EV_SEND_DMA_READY  
        Slave has loaded its TX data buffer to the hardware (DMA).

    enumerator SPI_EV_SEND  
        Master has received certain number of the data, the number is determined by Master.
enumerator **SPI_EV_RECV_DMA_READY**

Slave has loaded its RX data buffer to the hardware (DMA).

enumerator **SPI_EV_RECV**

Slave has received certain number of data from master, the number is determined by Master.

enumerator **SPI_EV_CMD9**

Received CMD9 from master.

enumerator **SPI_EV_CMDA**

Received CMDA from master.

enumerator **SPI_EV_TRANS**

A transaction has done.

enum **spi_command_t**

SPI command.

*Values:*

enumerator **SPI_CMD_HD_WRBUF**

enumerator **SPI_CMD_HD_RDBUF**

enumerator **SPI_CMD_HD_WRDMA**

enumerator **SPI_CMD_HD_RDDMA**

enumerator **SPI_CMD_HD_SEG_END**

enumerator **SPI_CMD_HD_EN_QPI**

enumerator **SPI_CMD_HD_WR_END**

enumerator **SPI_CMD_HD_INT0**

enumerator **SPI_CMD_HD_INT1**

enumerator **SPI_CMD_HD_INT2**

**Header File**

- components/esp_driver_spi/include/driver/spi_common.h
  - This header file can be included with:
    
    ```
    #include "driver/spi_common.h"
    ```

- This header file is a part of the API provided by the esp_driver_spi component. To declare that your component depends on esp_driver_spi, add the following to your CMakeLists.txt:
REQUIRES esp_driver_spi

or

PRIV_REQUIRES esp_driver_spi

Functions

esp_err_t spi_bus_initialize (spi_host_device_t host_id, const spi_bus_config_t *bus_config, spi_dma_chan_t dma_chan)

Initialize a SPI bus.

Warning: SPI0/1 is not supported

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.

Parameters

• host_id – 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 – Selecting a DMA channel for an SPI bus allows transactions on the bus with size only limited by the amount of internal memory.
  – Selecting SPI_DMA_DISABLED limits the size of transactions.
  – Set to SPI_DMA_DISABLED if only the SPI flash uses this bus.
  – Set to SPI_DMA_CH_AUTO to let the driver to allocate the DMA channel.

Returns

• ESP_ERR_INVALID_ARG if configuration is invalid
• ESP_ERR_INVALID_STATE if host already is in use
• ESP_ERR_NOT_FOUND if there is no available DMA channel
• ESP_ERR_NO_MEM if out of memory
• ESP_OK on success

esp_err_t spi_bus_free (spi_host_device_t host_id)

Free a SPI bus.

Warning: In order for this to succeed, all devices have to be removed first.

Parameters host_id – SPI peripheral to free

Returns

• ESP_ERR_INVALID_ARG if parameter is invalid
• ESP_ERR_INVALID_STATE if bus hasn’t been initialized before, or not all devices on the bus are freed
• ESP_OK on success

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 refering 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 data0_io_num
    GPIO pin for spi data0 signal in quad/octal mode, 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 data1_io_num
    GPIO pin for spi data1 signal in quad/octal mode, 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, or -1 if not used.

int data2_io_num
    GPIO pin for spi data2 signal in quad/octal mode, or -1 if not used.

int quadhd_io_num
    GPIO pin for HD (Hold) signal, or -1 if not used.

int data3_io_num
    GPIO pin for spi data3 signal in quad/octal mode, or -1 if not used.

int data4_io_num
    GPIO pin for spi data4 signal in octal mode, or -1 if not used.

int data5_io_num
    GPIO pin for spi data5 signal in octal mode, or -1 if not used.

int data6_io_num
    GPIO pin for spi data6 signal in octal mode, or -1 if not used.
int `data7_io_num`
   GPIO pin for spi data7 signal in octal mode, 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.

`esp_intr_cpu_affinity / isr_cpu_id`
   Select cpu core to register SPI ISR.

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:

   ```c
   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:

   ```c
   uint16_t data = SPI_SWAP_DATA_RX((uint32_t*)tx->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_GPIO_PINS
Forcing the signals to be routed through GPIO matrix. Or indicates the pins are routed through the 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 MOSI 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_IO4_IO7
Check existing of IO4-IO7 pins. Or indicates IO4-IO7 pins initialized.

SPICOMMON_BUSFLAG_OCTAL
Check existing of MOSI/MISO/WP/HD/SPIIO4/SPIIO5/SPIIO6/SPIIO7 pins as output. Or indicates bus able to work under octal mode.

SPICOMMON_BUSFLAG_NATIVE_PINS

Type Definitions
typedef spi_common_dma_t spi_dma_chan_t

Enumerations
enum spi_common_dma_t
SPI DMA channels.

Values:

enumerator SPI_DMA_DISABLED
Do not enable DMA for SPI.

denominator SPI_DMA_CH_AUTO
Enable DMA, channel is automatically selected by driver.
API Reference - SPI Master

Header File

- components/esp_driver_spi/include/driver/spi_master.h
- This header file can be included with:

```c
#include "driver/spi_master.h"
```

- This header file is a part of the API provided by the esp_driver_spi component. To declare that your component depends on esp_driver_spi, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_spi
```

or

```cmake
PRIV_REQUIRES esp_driver_spi
```

Functions

```c
esp_err_t spi_bus_add_device(spi_host_device_t host_id, 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.

**Parameters**

- `host_id` - SPI peripheral to allocate device on
- `dev_config` - SPI interface protocol config for the device
- `handle` - Pointer to variable to hold the device handle

**Returns**

- ESP_ERR_INVALID_ARG if parameter is invalid or configuration combination is not supported (e.g. `dev_config->post_cb` isn’t set while flag `SPI_DEVICE_NO_RETURN_RESULT` is enabled)
- ESP_ERR_INVALID_STATE if selected clock source is unavailable or spi bus not initialized
- 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

```c
esp_err_t spi_bus_remove_device(spi_device_handle_t handle)
```

Remove a device from the SPI bus.

**Parameters** `handle` - Device handle to free

**Returns**

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if device already is freed
- ESP_OK on success

```c
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`.

**Parameters**

- `handle` - SPI device handle
- `trans_desc` - SPI transaction descriptor
- `ticks_to_wait` - Number of ticks to wait for the transaction to complete

**Returns**

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_INVALID_STATE if device already is freed
- ESP_OK on success
**Chapter 2. API Reference**

**Note:** Normally a device cannot start (queue) polling and interrupt transactions simultaneously.

### 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.

### Returns
- **ESPR_ERR_INVALID_ARG** if parameter is invalid. This can happen if `SPI_TRANS_CS_KEEP_ACTIVE` flag is specified while the bus was not acquired (call `spi_device_acquire_bus` first) or set flag `SPI_TRANS_DMA_BUFFER_ALIGN_MANUAL` but tx or rx buffer is not DMA-capable, or `addr` and `len` do not align to cache line size
- **ESPR_ERR_TIMEOUT** if there was no room in the queue before `ticks_to_wait` expired
- **ESPR_ERR_NO_MEM** if allocating DMA-capable temporary buffer failed
- **ESPR_ERR_INVALID_STATE** if previous transactions are not finished
- **ESPR_OK** on success

```c
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.

### 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.

### Returns
- **ESPR_ERR_INVALID_ARG** if parameter is invalid
- **ESPR_ERR_NOT_SUPPORTED** if flag `SPI_DEVICE_NO_RETURN_RESULT` is set
- **ESPR_ERR_TIMEOUT** if there was no completed transaction before `ticks_to_wait` expired
- **ESPR_OK** on success

```c
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.

### Parameters
- **handle**: Device handle obtained using `spi_host_add_dev`
- **trans_desc**: Description of transaction to execute

### Returns
- **ESPR_ERR_INVALID_ARG** if parameter is invalid
- **ESPR_OK** on success
**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.

**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.

**Returns**
- `ESP_ERR_INVALID_ARG` if parameter is invalid. This can happen if `SPI_TRANS_CS_KEEP_ACTIVE` flag is specified while the bus was not acquired (`spi_device_acquire_bus()` should be called first) or set flag `SPI_TRANS_DMA_BUFFER_ALIGN_MANUAL` but tx or rx buffer not DMA-capable, or addr&len not align to cache line size
- `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

**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.

**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.

**Returns**
- `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

**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.
• ESP_OK on success

```c
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.

**Parameters**

- **device** - The device to occupy the bus.
- **wait** - Time to wait before the bus is occupied by the device. Currently MUST set to `portMAX_DELAY`.

**Returns**

- ESP_ERR_INVALID_ARG: `wait` is not set to `portMAX_DELAY`.
- ESP_OK: Success.

```c
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.

```c
esp_err_t spi_device_get_actual_freq(spi_device_handle_t handle, int* freq_khz)
```

Calculate working frequency for specific device.

**Parameters**

- **handle** - SPI device handle
- **freq_khz** [out] output parameter to hold calculated frequency in kHz

**Returns**

- ESP_ERR_INVALID_ARG: handle or `freq_khz` parameter is NULL
- ESP_OK: Success

```c
int spi_get_actual_clock(int fapb, int hz, int duty_cycle)
```

Calculate the working frequency that is most close to desired frequency.

**Parameters**

- **fapb** - The frequency of apb clock, should be `APB_CLK_FREQ`.
- **hz** - Desired working frequency
- **duty_cycle** - Duty cycle of the spi clock

**Returns**

Actual working frequency that most fit.

```c
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_get_actual_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.

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.

Returns Frequency limit of current configurations.

esp_err_t spi_bus_get_max_transaction_len (spi_host_device_t host_id, size_t *max_bytes)

Get max length (in bytes) of one transaction.

Parameters
- host_id - SPI peripheral
- max_bytes - [out] Max length of one transaction, in bytes

Returns
- ESP_OK: On success
- ESP_ERR_INVALID_ARG: Invalid argument

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, representing a pair of (CPOL, CPHA) configuration:
- 0: (0, 0)
- 1: (0, 1)
- 2: (1, 0)
- 3: (1, 1)

spi_clock_source_t clock_source
Select SPI clock source, SPI_CLK_SRC_DEFAULT by default.

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.
Chapter 2. API Reference

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
SPI clock speed in Hz. Derived from clock_source.

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.
Chapter 2. API Reference

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-IDF 3.0.
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.


```c
    uint8_t dummy_bits
    The dummy length in this transaction, in bits.
```

**Macros**

**SPI_MASTER_FREQ_8M**

SPI common used frequency (in Hz)

---

**Note:** SPI peripheral only has an integer divider, and the default clock source can be different on other targets, so the actual frequency may be slightly different from the desired frequency. 8MHz

---

**SPI_MASTER_FREQ_9M**

8.89MHz

**SPI_MASTER_FREQ_10M**

10MHz

**SPI_MASTER_FREQ_11M**

11.43MHz

**SPI_MASTER_FREQ_13M**

13.33MHz

**SPI_MASTER_FREQ_16M**

16MHz

**SPI_MASTER_FREQ_20M**

20MHz

**SPI_MASTER_FREQ_26M**

26.67MHz

**SPI_MASTER_FREQ_40M**

40MHz

**SPI_MASTER_FREQ_80M**

80MHz

**SPI_DEVICE_TXBIT_LSBFIRST**

Transmit command/address/data LSB first instead of the default MSB first.

**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_DEVICE_NO_RETURN_RESULT
Don’t return the descriptor to the host on completion (use post_cb to notify instead)

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_CS_KEEP_ACTIVE
Keep CS active after data transfer.

SPI_TRANS_MULTILINE_CMD
The data lines used at command phase is the same as data phase (otherwise, only one data line is used at
command phase)

SPI_TRANS_MODE_OCT
Transmit/receive data in 8-bit mode.

SPI_TRANS_MULTILINE_ADDR
The data lines used at address phase is the same as data phase (otherwise, only one data line is used at address
phase)

SPI_TRANS_DMA_BUFFER_ALIGN_MANUAL
By default driver will automatically re-alloc dma buffer if it doesn’t meet hardware alignment or dma_capable
requirements, this flag is for you to disable this feature, you will need to take care of the alignment otherwise
driver will return you error ESP_ERR_INVALID_ARG.

Type Definitions
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.

2.5.21 SPI Slave Driver

SPI Slave driver is a program that controls ESP32-S2’s General Purpose SPI (GP-SPI) peripheral(s) when it functions
as a slave.

For more hardware information about the GP-SPI peripheral(s), see ESP32-S2 Technical Reference Manual >
SPI Controller [PDF].

Terminology
The terms used in relation to the SPI slave driver are given in the table below.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>The SPI controller peripheral external to ESP32-S2 that initiates SPI transmissions over the bus, and acts as an SPI Master.</td>
</tr>
<tr>
<td>Device</td>
<td>SPI slave device (general purpose SPI controller). 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.</td>
</tr>
<tr>
<td>Bus</td>
<td>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. 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.</td>
</tr>
<tr>
<td>MISO</td>
<td>Master In, Slave Out, a.k.a. Q. Data transmission from a Device to Host.</td>
</tr>
<tr>
<td>MOSI</td>
<td>Master Out, Slave In, a.k.a. D. Data transmission from a Host to Device.</td>
</tr>
<tr>
<td>SCLK</td>
<td>Serial Clock. Oscillating signal generated by a Host that keeps the transmission of data bits in sync.</td>
</tr>
<tr>
<td>CS</td>
<td>Chip Select. Allows a Host to select individual Device(s) connected to the bus in order to send or receive data.</td>
</tr>
<tr>
<td>QUADWP</td>
<td>Write Protect signal. Only used for 4-bit (qio/qout) transactions.</td>
</tr>
<tr>
<td>QUADHD</td>
<td>Hold signal. Only used for 4-bit (qio/qout) transactions.</td>
</tr>
<tr>
<td>Assertion</td>
<td>The action of activating a line. The opposite action of returning the line back to inactive (back to idle) is called de-assertion.</td>
</tr>
<tr>
<td>Transaction</td>
<td>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.</td>
</tr>
<tr>
<td>Launch Edge</td>
<td>Edge of the clock at which the source register launches the signal onto the line.</td>
</tr>
<tr>
<td>Latch Edge</td>
<td>Edge of the clock at which the destination register latches in the signal.</td>
</tr>
</tbody>
</table>

**Driver Features**

The SPI slave driver allows using the SPI peripherals as full-duplex Devices. The driver can send/receive transactions up to 72 bytes in length, or utilize DMA to send/receive longer transactions. However, there are some known issues related to DMA.

The SPI slave driver supports registering the SPI ISR to a certain CPU core. If multiple tasks try to access the same SPI Device simultaneously, it is recommended that your application be refactored so that each SPI peripheral is only accessed by a single task at a time. Please also use `spi_bus_config_t::isr_cpu_id` to register the SPI ISR to the same core as SPI peripheral related tasks to ensure thread safety.

**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 peripheral 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 can choose 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. Similarly, 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`.  

Submit Document Feedback
Chapter 2. API Reference

Driver Usage

- Initialize an SPI peripheral as a Device by calling the function `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 by setting the parameter `dma_chan` to the host device. 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 `spi_slave_transaction_t::rx_buffer` and `spi_slave_transaction_t::tx_buffer`. 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 `spi_slave_transaction_t::length`. However, this member does not define the actual length of an SPI transaction. A transaction’s length is determined by the clock and CS lines driven by the Host. 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 `spi_slave_transaction_t::length` member will be sent and received. In this case, `spi_slave_transaction_t::trans_len` is set to `spi_slave_transaction_t::length` instead of the actual transaction length. To meet the actual transaction length requirements, set `spi_slave_transaction_t::trans_len` to a value greater than the maximum `spi_slave_transaction_t::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.

GPIO Matrix and IO_MUX

Most of chip’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.

When an SPI Host is set to 80 MHz or lower frequencies, routing SPI pins via GPIO matrix will behave the same compared to routing them via IO_MUX.

The IO_MUX pins for SPI buses are given below.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>GPIO Number (SPI2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS0</td>
<td>10</td>
</tr>
<tr>
<td>SCLK</td>
<td>12</td>
</tr>
<tr>
<td>MISO</td>
<td>13</td>
</tr>
<tr>
<td>MOSI</td>
<td>11</td>
</tr>
<tr>
<td>QUADWP</td>
<td>14</td>
</tr>
<tr>
<td>QUADHD</td>
<td>9</td>
</tr>
</tbody>
</table>

Speed and Timing Considerations

Transaction Interval

The ESP32-S2 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 40 MHz. The data cannot be recognized or received correctly if the clock is too fast or does not have a 50% duty cycle.

**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.

**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**

- `components/esp_driver_spi/include/driver/spi_slave.h`
- This header file can be included with:

```c
#include "driver/spi_slave.h"
```

- This header file is a part of the API provided by the `esp_driver_spi` component. To declare that your component depends on `esp_driver_spi`, add the following to your `CMakeLists.txt`:

```cmake
REQUIRES esp_driver_spi
```

or

```cmake
PRIV_REQUIRES esp_driver_spi
```

**Functions**

```c
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, spi_dma_chan_t dma_chan)
```

Initialize a SPI bus as a slave interface.

**Warning:** SPI0/1 is not supported

**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.

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** - Selecting a DMA channel for an SPI bus allows transactions on the bus with size only limited by the amount of internal memory.
  - Selecting SPI_DMA_DISABLED limits the size of transactions.
  - Set to SPI_DMA_DISABLED if only the SPI flash uses this bus.
  - Set to SPI_DMA_CH_AUTO to let the driver to allocate the DMA channel.

Returns

- ESP_ERR_INVALID_ARG if configuration is invalid
- ESP_ERR_INVALID_STATE if host already is in use
- ESP_ERR_NOT_FOUND if there is no available DMA channel
- ESP_ERR_NO_MEM if out of memory
- ESP_OK on success

```c
esp_err_t spi_slave_free (spi_host_device_t host)
```
Free a SPI bus claimed as a SPI slave interface.

Parameters **host** – SPI peripheral to free

Returns

- 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

```c
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.

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 port-MAX_DELAY to never time out.

Returns

- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NO_MEM if set flag `SPI_SLAVE_TRANS_DMA_BUFFER_ALIGN_AUTO` but there is no free memory
- ESP_ERR_INVALID_STATE if sync data between Cache and memory failed
- ESP_OK on success

```c
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`.

**Parameters**
- `host` - SPI peripheral to that is acting as a slave
- `trans_desc` - [out] 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.

**Returns**
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_ERR_NOT_SUPPORTED if flag `SPI_SLAVE_NO_RETURN_RESULT` is set
- ESP_OK on success

```c
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`.

**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.

**Returns**
- ESP_ERR_INVALID_ARG if parameter is invalid
- ESP_OK on success

**Structures**

```c
struct spi_slave_interface_config_t
```

This is a configuration for a SPI host acting as a slave device.

**Public Members**

```c
int spics_io_num
```

CS GPIO pin for this device.

```c
uint32_t flags
```

Bitwise OR of `SPI_SLAVE_*` flags.

```c
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.

```c
uint8_t mode
```

SPI mode, representing a pair of (CPOL, CPHA) configuration:
- 0: (0, 0)
- 1: (0, 1)
Chapter 2. API Reference

• 2: (1, 0)
• 3: (1, 1)

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

uint32_t flags
Bitwise OR of SPI_SLAVE_TRANS_* flags.

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.
**SPI_SLAVE_NO_RETURN_RESULT**

Don’t return the descriptor to the host on completion (use `post_trans_cb` to notify instead)

**SPI_SLAVE_TRANS_DMA_BUFFER_ALIGN_AUTO**

Automatically re-malloc dma buffer if user buffer doesn’t meet hardware alignment or `dma_capable`, this process may loss some memory and performance.

**Type Definitions**

typedef void (\*slave_transaction_cb_t)(spi_slave_transaction_t \*trans)

### 2.5.22 SPI Slave Half Duplex

**Introduction**

The Half Duplex (HD) Mode is a special mode provided by ESP SPI Slave peripheral. Under this mode, the hardware provides more services than the Full Duplex (FD) Mode (the mode for general-purpose SPI transactions, see **SPI Slave Driver**). These services reduce the CPU load and the response time of SPI Slave. However, it is important to note that the communication format is determined by the hardware and is always in a half-duplex configuration, allowing only one-way data transfer at any given time. Hence, the mode is named Half Duplex Mode due to this characteristic.

When conducting an SPI transaction, transactions can be classified into several types based on the **command** phase of the transaction. Each transaction may consist of the following phases: command, address, dummy, and data. The command phase is mandatory, while the other phases may be determined by the command field. During the command, address, and dummy phases, the bus is always controlled by the master (usually the host), while the direction of the data phase depends on the command. The data phase can be either an input phase, where the master writes data to the slave (e.g., the host sends data to the slave), or an output phase, where the master reads data from the slave (e.g., the host receives data from the slave).

**Protocol** About the details of how master should communicate with the SPI Slave, see **ESP SPI Slave HD (Half Duplex) Mode Protocol**.

Through these different transactions, the slave provides these services to the master:

- A DMA channel for the master to write a great amount of data to the slave.
- A DMA channel for the master to read a great amount of data from the slave.
- Several general purpose registers, shared between the master and the slave.
- Several general purpose interrupts, for the master to interrupt the SW of the slave.

**Terminology**

- Transaction
- Channel
- Sending
- Receiving
- Data Descriptor

**Driver Feature**

- Transaction read/write by master in segments
- Queues for data to send and received
Driver Usage

**Slave Initialization** Call `spi_slave_hd_init()` to initialize the SPI bus as well as the peripheral and the driver. The SPI Slave exclusively uses the SPI peripheral, pins of the bus before it is deinitialized, which means other devices are unable to use the above resources during initialization. Thus, to ensure SPI resources are correctly occupied and the connections work properly, most configurations of the slave should be done as soon as the slave is initialized.

The `spi_bus_config_t` specifies how the bus should be initialized, while `spi_slave_hd_slot_config_t` specifies how the SPI Slave driver should work.

**Deinitialization (Optional)** Call `spi_slave_hd_deinit()` to uninstall the driver. The resources, including the pins, SPI peripheral, internal memory used by the driver, and interrupt sources, are released by the `deinit()` function.

**Send/Receive Data by DMA Channels** To send data to the master through the sending DMA channel, the application should properly wrap the data in an `spi_slave_hd_data_t` descriptor structure before calling `spi_slave_hd_queue_trans()` with the data descriptor and the channel argument of `SPI_SLAVE_CHAN_TX`. The pointers to descriptors are stored in the queue, and the data is sent to the master in the same order they are enqueued using `spi_slave_hd_queue_trans()`, upon receiving the master’s `Rd_DMA` command.

The application should check the result of data sending by calling `spi_slave_hd_get_trans_res()` with the channel set as `SPI_SLAVE_CHAN_TX`. This function blocks until the transaction with the command `Rd_DMA` from the master successfully completes (or timeout). The `out_trans` argument of the function outputs the pointer of the data descriptor which is just finished, providing information about the sending.

Receiving data from the master through the receiving DMA channel is quite similar. The application calls `spi_slave_hd_queue_trans()` with proper data descriptor and the channel argument of `SPI_SLAVE_CHAN_RX`. And the application calls the `spi_slave_hd_get_trans_res()` later to get the descriptor to the receiving buffer before it handles the data in the receiving buffer.

**Note:** This driver itself does not have an internal buffer for the data to send or just received. The application should provide data buffer for driver via data descriptors to send to the master, or to receive data from the master.

The application has to properly keep the data descriptor as well as the buffer it points, after the descriptor is successfully sent into the driver internal queue by `spi_slave_hd_queue_trans()`, and before returned by `spi_slave_hd_get_trans_res()`. During this period, the hardware as well as the driver may read or write to the buffer and the descriptor when required at any time.

Please note that, when using this driver for data transfer, the buffer does not have to be fully sent or filled before it is terminated. For example, in the segment transaction mode, the master has to send `CMD7` to terminate a `Wr_DMA` transaction or send `CMD8` to terminate an `Rd_DMA` transaction (in segments), no matter whether the send (receive) buffer is used up (full) or not.

**Using Data Descriptor with Customized User Arguments** Sometimes you may have initiator (sending data descriptor) and closure (handling returned descriptors) functions in different places. When you get the returned data descriptor in the closure, you may need some extra information when handling the finished data descriptor. For example, you may want to know which round it is for the returned descriptor when you send the same piece of data several times.

Set the `arg` member in the data descriptor to a variable indicating the transaction by force casting, or point it to a structure that wraps all the information you may need when handling the sending/receiving data. Then you can get what you need in your closure.
**Using Callbacks**

**Note:** These callbacks are called in the ISR, so the required operations need to be processed quickly and returned as soon as possible to ensure that the system is functioning properly. You may need to be very careful to write the code in the ISR.

Since the interrupt handling is executed concurrently with the application, long delays or blocking may cause the system to respond slower or lead to unpredictable behavior. Therefore, when writing callback functions, avoid using operations that may cause delays or blocking, e.g., waiting, sleeping, resource locking, etc.

The `spi_slave_hd_callback_config_t` member in the `spi_slave_hd_slot_config_t` configuration structure passed when initializing the SPI Slave HD driver, allows you to have callbacks for each event you may concern.

The corresponding interrupt for each callback that is not `NULL` is enabled, so that the callbacks can be called immediately when the events happen. You do not need to provide callbacks for the unconcerned events.

The `arg` member in the configuration structure can help you pass some context to the callback or indicate the specific SPI Slave instance when using the same callbacks for multiple SPI Slave peripherals. You can set the `arg` member to a variable that indicates the SPI Slave instance by performing a forced type casting or point it to a context structure. All the callbacks are called with this `arg` argument you set when the callbacks are initialized.

There are two other arguments: the `event` and the `awoken`.

- The `event` passes the information of the current event to the callback. The `spi_slave_hd_event_t` type contains the information of the event, for example, event type, the data descriptor just finished (The `data argument` is very useful in this case!).
- The `awoken` argument serves as an output parameter. It informs the ISR that tasks have been awakened after the callback function, and the ISR should call `portYIELD_FROM_ISR()` to schedule these tasks. Simply pass the `awoken` argument to all FreeRTOS APIs that may unblock tasks, and the value of `awoken` will be returned to the ISR.

**Writing/Reading Shared Registers**  Call `spi_slave_hd_write_buffer()` to write the shared buffer, and `spi_slave_hd_read_buffer()` to read the shared buffer.

**Note:** On ESP32-S2, the shared registers are read/written in words by the application but read/written in bytes by the master. There is no guarantee four continuous bytes read from the master are from the same word written by the slave’s application. It is also possible that if the slave reads a word while the master is writing bytes of the word, the slave may get one word with half of them just written by the master, and the other half has not been written into.

The master can confirm that the word is not in transition by reading the word twice and comparing the values.

For the slave, it is more difficult to ensure the word is not in transition because the process of master writing four bytes can be very long (32 SPI clocks). You can put some CRC in the last (largest address) byte of a word so that when the byte is written, the word is sure to be all written.

Due to the conflicts that may be among read/write from SW (worse if there are multi-cores) and master, it is suggested that a word is only used in one direction (only written by the master or only written by the slave).

**Receiving General Purpose Interrupts from the Master**  When the master sends `CMD8`, `CMD9` or `CMDA`, the slave corresponding is triggered. Currently the `CMD8` is permanently used to indicate the termination of `Rd_DMA` segments. To receive general-purpose interrupts, register callbacks for `CMD9` and `CMDA` when the slave is initialized, see **Using Callbacks**.

**Application Example**

The code example for Device/Host communication can be found in the `peripherals/spi_slave_hd` directory of ESP-IDF examples.
API Reference

Header File

- components/esp_driver_spi/include/driver/spi_slave_hd.h
- This header file can be included with:

```c
#include "driver/spi_slave_hd.h"
```
- This header file is a part of the API provided by the esp_driver_spi component. To declare that your component depends on esp_driver_spi, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_spi
```

or

```cmake
PRIV_REQUIRES esp_driver_spi
```

Functions

**esp_err_t spi_slave_hd_init (spi_host_device_t host_id, const spi_bus_config_t *bus_config, const spi_slave_hd_slot_config_t *config)**

Initialize the SPI Slave HD driver.

**Parameters**
- `host_id` - The host to use
- `bus_config` - Bus configuration for the bus used
- `config` - Configuration for the SPI Slave HD driver

**Returns**
- `ESP_OK`: on success
- `ESP_ERR_INVALID_ARG`: invalid argument given
- `ESP_ERR_INVALID_STATE`: function called in invalid state, may be some resources are already in use
- `ESP_ERR_NOT_FOUND` if there is no available DMA channel
- `ESP_ERR_NO_MEM`: memory allocation failed
- or other return value from esp_intr_alloc

**esp_err_t spi_slave_hd_deinit (spi_host_device_t host_id)**

Deinitialize the SPI Slave HD driver.

**Parameters**
- `host_id` - The host to deinitialize the driver

**Returns**
- `ESP_OK`: on success
- `ESP_ERR_INVALID_ARG`: if the host_id is not correct

**esp_err_t spi_slave_hd_queue_trans (spi_host_device_t host_id, spi_slave_chan_t chan, spi_slave_hd_data_t *trans, TickType_t timeout)**

Queue transactions (segment mode)

**Parameters**
- `host_id` - Host to queue the transaction
- `chan` - SPI_SLAVE_CHAN_TX or SPI_SLAVE_CHAN_RX
- `trans` - Transaction descriptors
- `timeout` - Timeout before the data is queued

**Returns**
- `ESP_OK`: on success
- `ESP_ERR_INVALID_ARG`: The input argument is invalid. Can be the following reason:
  - The buffer given is not DMA capable
  - The length of data is invalid (not larger than 0, or exceed the max transfer length)
  - The transaction direction is invalid
- `ESP_ERR_TIMEOUT`: Cannot queue the data before timeout. Master is still processing previous transaction.
• ESP_ERR_INVALID_STATE: Function called in invalid state. This API should be called under segment mode.

```c
esp_err_t spi_slave_hd_get_trans_res(spi_host_device_t host_id, spi_slave_chan_t chan, spi_slave_hd_data_t **out_trans, TickType_t timeout)
```

Get the result of a data transaction (segment mode)

**Note:** This API should be called successfully the same times as the `spi_slave_hd_queue_trans`.

### Parameters

- **host_id** – Host to queue the transaction
- **chan** – Channel to get the result, SPI_SLAVE_CHAN_TX or SPI_SLAVE_CHAN_RX
- **out_trans** – [out] Pointer to the transaction descriptor (`spi_slave_hd_data_t`) passed to the driver before. Hardware has finished this transaction. Member `trans_len` indicates the actual number of bytes of received data, it’s meaningless for TX.
- **timeout** – Timeout before the result is got

### Returns

- ESP_OK: on success
- ESP_ERR_INVALID_ARG: Function is not valid
- ESP_ERR_TIMEOUT: There’s no transaction done before timeout
- ESP_ERR_INVALID_STATE: Function called in invalid state. This API should be called under segment mode.

```c
void spi_slave_hd_read_buffer(spi_host_device_t host_id, int addr, uint8_t *out_data, size_t len)
```

Read the shared registers.

### Parameters

- **host_id** – Host to read the shared registers
- **addr** – Address of register to read, 0 to `SOC_SPI_MAXIMUM_BUFFER_SIZE-1`
- **out_data** – [out] Output buffer to store the read data
- **len** – Length to read, not larger than `SOC_SPI_MAXIMUM_BUFFER_SIZE-addr`

```c
void spi_slave_hd_write_buffer(spi_host_device_t host_id, int addr, uint8_t *data, size_t len)
```

Write the shared registers.

### Parameters

- **host_id** – Host to write the shared registers
- **addr** – Address of register to write, 0 to `SOC_SPI_MAXIMUM_BUFFER_SIZE-1`
- **data** – Buffer holding the data to write
- **len** – Length to write, `SOC_SPI_MAXIMUM_BUFFER_SIZE-addr`

```c
esp_err_t spi_slave_hd_append_trans(spi_host_device_t host_id, spi_slave_chan_t chan, spi_slave_hd_data_t *trans, TickType_t timeout)
```

Load transactions (append mode)

**Note:** In this mode, user transaction descriptors will be appended to the DMA and the DMA will keep processing the data without stopping

### Parameters

- **host_id** – Host to load transactions
- **chan** – SPI_SLAVE_CHAN_TX or SPI_SLAVE_CHAN_RX
- **trans** – Transaction descriptor
- **timeout** – Timeout before the transaction is loaded

### Returns

- ESP_OK: on success
- ESP_ERR_INVALID_ARG: The input argument is invalid. Can be the following reason:
– The buffer given is not DMA capable
– The length of data is invalid (not larger than 0, or exceed the max transfer length)
– The transaction direction is invalid
• ESP_ERR_TIMEOUT: Master is still processing previous transaction. There is no available transaction for slave to load
• ESP_ERR_INVALID_STATE: Function called in invalid state. This API should be called under append mode.

```c
esp_err_t spi_slave_hd_get_append_trans_res(spi_host_device_t host_id, spi_slave_chan_t chan, spi_slave_hd_data_t **out_trans, TickType_t timeout)
```

Get the result of a data transaction (append mode)

**Note:** This API should be called the same times as the `spi_slave_hd_append_trans`

### Parameters
- `host_id` – Host to load the transaction
- `chan` – SPI_SLAVE_CHAN_TX or SPI_SLAVE_CHAN_RX
- `out_trans` – [out] Pointer to the transaction descriptor (`spi_slave_hd_data_t`) passed to the driver before. Hardware has finished this transaction. Member `trans_len` indicates the actual number of bytes of received data, it’s meaningless for TX.
- `timeout` – Timeout before the result is got

### Returns
- ESP_OK: on success
- ESP_ERR_INVALID_ARG: Function is not valid
- ESP_ERR_TIMEOUT: There’s no transaction done before timeout
- ESP_ERR_INVALID_STATE: Function called in invalid state. This API should be called under append mode.

### Structures

```c
struct spi_slave_hd_data_t
```

Descriptor of data to send/receive.

#### Public Members

- `uint8_t *data`
  
  Buffer to send, must be DMA capable.

- `size_t len`
  
  Len of data to send/receive. For receiving the buffer length should be multiples of 4 bytes, otherwise the extra part will be truncated.

- `size_t trans_len`
  
  For RX direction, it indicates the data actually received. For TX direction, it is meaningless.

- `uint32_t flags`
  
  Bitwise OR of SPI_SLAVE_HD_TRANS_* flags.

- `void *arg`
  
  Extra argument indicating this data.
struct `spi_slave_hd_event_t`

Information of SPI Slave HD event.

**Public Members**

- `spi_event_t event`
  Event type.

- `spi_slave_hd_data_t *trans`
  Corresponding transaction for SPI_EV_SEND and SPI_EV_RECV events.

struct `spi_slave_hd_callback_config_t`

Callback configuration structure for SPI Slave HD.

**Public Members**

- `slave_cb_t cb_buffer_tx`
  Callback when master reads from shared buffer.

- `slave_cb_t cb_buffer_rx`
  Callback when master writes to shared buffer.

- `slave_cb_t cb_send_dma_ready`
  Callback when TX data buffer is loaded to the hardware (DMA)

- `slave_cb_t cb_sent`
  Callback when data are sent.

- `slave_cb_t cb_recv_dma_ready`
  Callback when RX data buffer is loaded to the hardware (DMA)

- `slave_cb_t cb_recv`
  Callback when data are received.

- `slave_cb_t cb_cmd9`
  Callback when CMD9 received.

- `slave_cb_t cb_cmdA`
  Callback when CMDA received.

- `void *arg`
  Argument indicating this SPI Slave HD peripheral instance.

struct `spi_slave_hd_slot_config_t`

Configuration structure for the SPI Slave HD driver.
Chapter 2. API Reference

Public Members

`uint8_t mode`
SPI mode, representing a pair of (CPOL, CPHA) configuration:
- 0: (0, 0)
- 1: (0, 1)
- 2: (1, 0)
- 3: (1, 1)

`uint32_t spics_io_num`
CS GPIO pin for this device.

`uint32_t flags`
Bitwise OR of SPI_SLAVE_HD_* flags.

`uint32_t command_bits`
command field bits, multiples of 8 and at least 8.

`uint32_t address_bits`
address field bits, multiples of 8 and at least 8.

`uint32_t dummy_bits`
dummy field bits, multiples of 8 and at least 8.

`uint32_t queue_size`
Transaction queue size. This sets how many transactions can be ‘in the air’ (queued using spi_slave_hd_queue_trans but not yet finished using spi_slave_hd_get_trans_result) at the same time.

`spi_dma_chan_t dma_chan`
DMA channel to used.

`spi_slave_hd_callback_config_t cb_config`
Callback configuration.

Macros

`SPI_SLAVE_HD_TRANS_DMA_BUFFER_ALIGN_AUTO`
Automatically re-malloc dma buffer if user buffer doesn’t meet hardware alignment or dma_capable, this process may lose some memory and performance.

`SPI_SLAVE_HD_TXBIT_LSBFIRST`
Transmit command/address/data LSB first instead of the default MSB first.

`SPI_SLAVE_HD_RXBIT_LSBFIRST`
Receive data LSB first instead of the default MSB first.

`SPI_SLAVE_HD_BIT_LSBFIRST`
Transmit and receive LSB first.
**SPI_SLAVE_HD_APPEND_MODE**

Adopt DMA append mode for transactions. In this mode, users can load(append) DMA descriptors without stopping the DMA.

**Type Definitions**

define bool (*slave_cb_t)(void *arg, spi_slave_hd_event_t *event, BaseType_t *awoken)

Callback for SPI Slave HD.

**Enumerations**

enum spi_slave_chan_t

Channel of SPI Slave HD to do data transaction.

Values:

enumerator SPI_SLAVE_CHAN_TX

The output channel (RDDMA)

enumerator SPI_SLAVE_CHAN_RX

The input channel (WRDMA)

### 2.5.23 Temperature Sensor

**Introduction**

The ESP32-S2 has a built-in sensor used to measure the chip’s internal temperature. The temperature sensor module contains an 8-bit Sigma-Delta analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) to compensate for the temperature measurement.

Due to restrictions of hardware, the sensor has predefined measurement ranges with specific measurement errors. See the table below for details.

<table>
<thead>
<tr>
<th>Predefined Range (°C)</th>
<th>Error (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 ~ 125</td>
<td>&lt; 3</td>
</tr>
<tr>
<td>20 ~ 100</td>
<td>&lt; 2</td>
</tr>
<tr>
<td>-10 ~ 80</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>-30 ~ 50</td>
<td>&lt; 2</td>
</tr>
<tr>
<td>-40 ~ 20</td>
<td>&lt; 3</td>
</tr>
</tbody>
</table>

**Note:** The temperature sensor is designed primarily to measure the temperature changes inside the chip. The internal temperature of a chip is usually higher than the ambient temperature, and is affected by factors such as the microcontroller’s clock frequency or I/O load, and the external thermal environment.

**Functional Overview**

The description of the temperature sensor functionality is divided into the following sections:
**Resource Allocation** - covers which parameters should be set up to get a temperature sensor handle and how to recycle the resources when the temperature sensor finishes working.

**Enable and Disable Temperature Sensor** - covers how to enable and disable the temperature sensor.

**Get Temperature Value** - covers how to get the real-time temperature value.

**Power Management** - covers how the temperature sensor is affected when changing power mode (e.g., Light-sleep mode).

**Thread Safety** - covers how to make the driver to be thread-safe.

**Resource Allocation**  
The ESP32-S2 has just one built-in temperature sensor hardware. The temperature sensor instance is represented by `temperature_sensor_handle_t`, which is also the bond of the context. By using `temperature_sensor_handle_t`, the temperature sensor properties can be accessed and modified in different function calls to control and manage the temperature sensor. The variable would always be the parameter of the temperature APIs with the information of hardware and configurations, so you can just create a pointer of type `temperature_sensor_handle_t` and passing to APIs as needed.

In order to install a built-in temperature sensor instance, the first thing is to evaluate the temperature range in your detection environment. For example, if the testing environment is in a room, the range you evaluate might be 10 °C ~ 30 °C; if the testing in a lamp bulb, the range you evaluate might be 60 °C ~ 110 °C. Based on that, configuration structure `temperature_sensor_config_t` should be defined in advance:

- `range_min`: The minimum value of the testing range you have evaluated.
- `range_max`: The maximum value of the testing range you have evaluated.

After the ranges are set, the structure could be passed to `temperature_sensor_install()`, which will instantiate the temperature sensor instance and return a handle.

As mentioned above, different measure ranges have different measurement errors. You do not need to care about the measurement error because we have an internal mechanism to choose the minimum error according to the given range.

If the temperature sensor is no longer needed, you need to call `temperature_sensor_uninstall()` to free the temperature sensor resource.

**Creating a Temperature Sensor Handle**

1. Step 1: Evaluate the testing range. In this example, the range is 20 °C ~ 50 °C.
2. Step 2: Configure the range and obtain a handle.

```c
    temperature_sensor_handle_t temp_handle = NULL;
    temperature_sensor_config_t temp_sensor = {
        .range_min = 20,
        .range_max = 50,
    };
    ESP_ERROR_CHECK(temperature_sensor_install(&temp_sensor, &temp_handle));
```

**Enable and Disable Temperature Sensor**

1. Enable the temperature sensor by calling `temperature_sensor_enable()`. The internal temperature sensor circuit will start to work. The driver state will transit from init to enable.
2. To Disable the temperature sensor, please call `temperature_sensor_disable()`.

**Get Temperature Value**  
After the temperature sensor is enabled by `temperature_sensor_enable()`, you can get the current temperature by calling `temperature_sensor_get_celsius()`.

```c
    // Enable temperature sensor
    ESP_ERROR_CHECK(temperature_sensor_enable(temp_handle));
    // Get converted sensor data
    float tsens_out;
    ESP_ERROR_CHECK(temperature_sensor_get_celsius(temp_handle, &tsens_out));
```

(continues on next page)
printf("Temperature in %f °C\n", tsens_out);

// Disable the temperature sensor if it is not needed and save the power
ESP_ERROR_CHECK(temperature_sensor_disable(temp_handle));

### Power Management

As the temperature sensor does not use the APB clock, it will keep working no matter if the power management is enabled with `CONFIG_PM_ENABLE`.

### Thread Safety

In the temperature sensor driver, we do not add any protection to ensure the thread safety, because typically this driver is only supposed to be used in one task. If you have to use this driver in different tasks, please add extra locks to protect it.

### Unexpected Behaviors

1. The value you get from the chip is usually different from the ambient temperature. It is because the temperature sensor is built inside the chip. To some extent, it measures the temperature of the chip.
2. When installing the temperature sensor, the driver may print the boundary you gave cannot meet the range of internal temperature sensor. It is because the built-in temperature sensor has a testing limit. The error comes from the incorrect configuration of `temperature_sensor_config_t` as follow:
   a) Totally out of range, like 200 °C ~ 300 °C.
   b) Cross the boundary of each predefined measurement. like 40 °C ~ 110 °C.

### Application Example

- Temperature sensor reading example: `peripherals/temperature_sensor/temp_sensor`

### API Reference

#### Header File

- `components/driver/temperature_sensor/include/driver/temperature_sensor.h`
  - This header file can be included with:

    ```c
    #include "driver/temperature_sensor.h"
    ```

  - This header file is a part of the API provided by the `driver` component. To declare that your component depends on `driver`, add the following to your CMakeLists.txt:

    ```cmake
    REQUIRE driver
    ```
    or
    ```cmake
    PRIV_REQUIRE driver
    ```

#### Functions

- `esp_err_t temperature_sensor_install` (const `temperature_sensor_config_t *tsens_config`, `temperature_sensor_handle_t *ret_tsens`)

  Install temperature sensor driver.

  **Parameters**
  - `tsens_config` – Pointer to config structure.
  - `ret_tsens` – Return the pointer of temperature sensor handle.

  **Returns**
Chapter 2. API Reference

- ESP_OK if succeed

```c
esp_err_t temperature_sensor_uninstall (temperature_sensor_handle_t tsens)
```
Uninstall the temperature sensor driver.

Parameters
- **tsens** - The handle created by `temperature_sensor_install()`.

Returns
- ESP_OK if succeed.

```c
esp_err_t temperature_sensor_enable (temperature_sensor_handle_t tsens)
```
Enable the temperature sensor.

Parameters
- **tsens** - The handle created by `temperature_sensor_install()`.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_STATE if temperature sensor is enabled already.

```c
esp_err_t temperature_sensor_disable (temperature_sensor_handle_t tsens)
```
Disable temperature sensor.

Parameters
- **tsens** - The handle created by `temperature_sensor_install()`.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_STATE if temperature sensor is not enabled yet.

```c
esp_err_t temperature_sensor_get_celsius (temperature_sensor_handle_t tsens, float *out_celsius)
```
Read temperature sensor data that is converted to degrees Celsius.

Note: Should not be called from interrupt.

Parameters
- **tsens** - The handle created by `temperature_sensor_install()`.
- **out_celsius** - The measure output value.

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG invalid arguments
- ESP_ERR_INVALID_STATE if temperature sensor is not enabled yet.
- ESP_FAIL Parse the sensor data into ambient temperature failed (e.g. out of the range).

Structures

```c
struct temperature_sensor_config_t
```
Configuration of measurement range for the temperature sensor.

Note: If you see the log the boundary you gave cannot meet the range of internal temperature sensor. You may need to refer to predefined range listed doc api-reference/peripherals/Temperature sensor.

Public Members

```c
int range_min
```
the minimum value of the temperature you want to test

```c
int range_max
```
the maximum value of the temperature you want to test
temperature_sensor_clk_src_t clk_src
the clock source of the temperature sensor.

Macros
TEMPERATURE_SENSOR_CONFIG_DEFAULT (min, max)

temperature_sensor_config_t default constructure

Type Definitions
typedef struct temperature_sensor_obj_t *temperature_sensor_handle_t
Type of temperature sensor driver handle.

2.5.24 Touch Sensor

Introduction

A touch sensor system is built on a substrate which carries electrodes and relevant connections under a protective flat surface. When the surface is touched, the capacitance variation is used to evaluate if the touch was valid.

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.

For design, operation, and control registers of a touch sensor, see ESP32-S2 Technical Reference Manual > On-Chip Sensors and Analog Signal Processing [PDF].

In-depth design details of touch sensors and firmware development guidelines for ESP32-S2 are available in Touch Sensor Application Note.

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 \texttt{touch_pad_config()}. The following 14 capacitive touch pads are supported for ESP32-S2.

<table>
<thead>
<tr>
<th>Touch Pad</th>
<th>GPIO Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>Internal channel, not connect to a GPIO</td>
</tr>
<tr>
<td>T1</td>
<td>GPIO1</td>
</tr>
<tr>
<td>T2</td>
<td>GPIO2</td>
</tr>
<tr>
<td>T3</td>
<td>GPIO3</td>
</tr>
<tr>
<td>T4</td>
<td>GPIO4</td>
</tr>
<tr>
<td>T5</td>
<td>GPIO5</td>
</tr>
<tr>
<td>T6</td>
<td>GPIO6</td>
</tr>
<tr>
<td>T7</td>
<td>GPIO7</td>
</tr>
<tr>
<td>T8</td>
<td>GPIO8</td>
</tr>
<tr>
<td>T9</td>
<td>GPIO9</td>
</tr>
<tr>
<td>T10</td>
<td>GPIO10</td>
</tr>
<tr>
<td>T11</td>
<td>GPIO11</td>
</tr>
<tr>
<td>T12</td>
<td>GPIO12</td>
</tr>
<tr>
<td>T13</td>
<td>GPIO13</td>
</tr>
<tr>
<td>T14</td>
<td>GPIO14</td>
</tr>
</tbody>
</table>

Use the function \texttt{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 \texttt{touch_pad_sw_start()} to start the FSM.

**Touch State Measurements**  The following function come in handy to read raw measurements from the sensor:

- \texttt{touch_pad_read_raw_data()}

It 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.

For the demonstration of how to read the touch pad data, check the application example \texttt{peripherals/touch_sensor/touch_sensor_v2/touch_pad_read}.

**Method of Measurements**  The touch sensor records the period of time (i.e., the number of clock cycles) over a fixed charge/discharge cycles (specified by \texttt{touch_pad_set_charge_discharge_times()}). The count result is the raw data that read from \texttt{touch_pad_read_raw_data()}. After finishing one measurement, the touch sensor sleeps until the next measurement start, this interval between two measurements can be set by \texttt{touch_pad_set_measurement_interval()}.

**Note:** If the specified charge and discharge cycles for measurement is too small, the result may be inaccurate, but increasing charge and discharge cycles will increase the power consumption as well. Additionally, the response of the touch sensor will slow down if the total time of the interval and measurement is too long.

**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 \texttt{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 \texttt{touch_pad_set_charge_discharge_times()}.

The following list summarizes available measurement parameters and corresponding ‘set’ functions:

- Touch pad charge / discharge parameters:
- voltage range: \texttt{touch_pad_set_voltage()}
- speed (slope): \texttt{touch_pad_set_cnt_mode()}

- Charge and discharge times of one measurement: \texttt{touch_pad_set_charge_discharge_times()}

Relationship between the voltage range (high/low reference voltages), speed (slope), and measurement time is shown in the figure below.

![Diagram](image.png)

The last chart \textbf{Output} represents the touch sensor reading, i.e., the time taken to accumulate the fixed number of cycles.

All functions are provided in pairs to set a specific parameter and to get the current parameter’s value, e.g., \texttt{touch_pad_set_voltage()} and \texttt{touch_pad_get_voltage()}.

**Filtering of Measurements** If measurements are noisy, you can filter them with provided API functions. The ESP32-S2’s touch functionality provides two sets of APIs for doing this.

There is an internal touch channel that is not connected to any external GPIO. The measurements from this denoise pad can be used to filter out interference introduced on all channels, such as noise introduced by the power supply and external EMI.

The denoise parameters are set with the function \texttt{touch_pad_denoise_set_config()} and started by calling \texttt{touch_pad_denoise_enable()}

There is also a configurable hardware implemented IIR-filter (infinite impulse response). This IIR-filter is configured with the function \texttt{touch_pad_filter_set_config()} and enabled by calling \texttt{touch_pad_filter_enable()}

**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 \texttt{touch_pad_get_status()} to check which pads have been touched and \texttt{touch_pad_clear_status()} to clear the touch status information.
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_sensor/touch_sensor_v2/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().

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().

**Application Examples**

- Touch sensor read example: peripherals/touch_sensor/touch_sensor_v2/touch_pad_read.
- Touch sensor interrupt example: peripherals/touch_sensor/touch_sensor_v2/touch_pad_interrupt.

**API Reference**

**Header File**

- components/driver/touch_sensor/esp32s2/include/driver/touch_sensor.h
  - This header file can be included with:
    ```
    #include "driver/touch_sensor.h"
    ```

  - This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
    ```
    REQUIRES driver
    ```
    or
    ```
    PRIV_REQUIRES driver
    ```

**Functions**

`esp_err_t touch_pad fsm_start (void)`

Set touch sensor FSM start.

**Note:** Start FSM after the touch sensor FSM mode is set.

**Note:** Call this function will reset benchmark of all touch channels.

**Returns**
Chapter 2. API Reference

- ESP_OK on success

```c
esp_err_t touch_pad_fsm_stop (void)
```

Stop touch sensor FSM.

**Returns**
- ESP_OK on success

```c
esp_err_t touch_pad_sw_start (void)
```

Trigger a touch sensor measurement, only support in SW mode of FSM.

**Returns**
- ESP_OK on success

```c
esp_err_t touch_pad_set_charge_discharge_times (uint16_t charge_discharge_times)
```

Set charge and discharge times of each measurement.

**Note:** This function will specify the charge and discharge times in each measurement period. The clock is sourced from SOC_MOD_CLK_RTC_FAST, and its default frequency is SOC_CLK_RC_FAST_FREQ_APPROX. The touch sensor will record the total clock cycles of all the charge and discharge cycles as the final result (raw value).

**Note:** If the charge and discharge times is too small, it may lead to inaccurate results.

**Parameters** `charge_discharge_times` - Charge and discharge times, range: 0 ~ 0xffff. No exact typical value can be recommended because the capacity is influenced by the hardware design and how finger touches, but suggest adjusting this value to make the measurement time around 1 ms.

**Returns**
- ESP_OK Set charge and discharge times success

```c
esp_err_t touch_pad_get_charge_discharge_times (uint16_t *charge_discharge_times)
```

Get charge and discharge times of each measurement.

**Parameters** `charge_discharge_times` - Charge and discharge times

**Returns**
- ESP_OK Get charge_discharge_times success
- ESP_ERR_INVALID_ARG The input parameter is NULL

```c
esp_err_t touch_pad_set_measurement_interval (uint16_t interval_cycle)
```

Set the interval between two measurements.

**Note:** The touch sensor will sleep between two measurements. This function is to set the interval cycle. And the interval is clocked from SOC_MOD_CLK_RTC_SLOW, its default frequency is SOC_CLK_RC_SLOW_FREQ_APPROX.

**Parameters** `interval_cycle` - The interval between two measurements sleep_time = interval_cycle / SOC_CLK_RC_SLOW_FREQ_APPROX. The approximate frequency value of RTC_SLOW_CLK can be obtained using rtc_clk_slow_freq_get_hz function.

**Returns**
- ESP_OK Set interval cycle success

```c
esp_err_t touch_pad_get_measurement_interval (uint16_t *interval_cycle)
```

Get the interval between two measurements.

**Parameters** `interval_cycle` - The interval between two measurements.
Returns
- ESP_OK Get interval cycle success
- ESP_ERR_INVALID_ARG The input parameter is NULL

`esp_err_t touch_pad_set_meas_time(uint16_t sleep_cycle, uint16_t meas_times)`

Set touch sensor times of charge and discharge and sleep time. Excessive total time will slow down the touch response. Too small measurement time will not be sampled enough, resulting in inaccurate measurements.

**Note:** The touch sensor will measure time of a fixed number of charge/discharge cycles (specified as the second parameter). That means the time (raw value) will increase as the capacity of the touch pad is increasing. The time (raw value) here is the number of clock cycles which is sourced from SOC_MOD_CLK_RTC_FAST and at (SOC_CLK_RC_FAST_FREQ_APPROX) Hz as default.

**Note:** The greater the duty cycle of the measurement time, the more system power is consumed.

**Parameters**
- `sleep_cycle` –The touch sensor will sleep after each measurement. `sleep_cycle` decide the interval between each measurement. `t_sleep = sleep_cycle / SOC_CLK_RC_SLOW_FREQ_APPROX`. The approximate frequency value of `RTC_SLOW_CLK` can be obtained using `rtc_clk_slow_freq_get_hz` function.
- `meas_times` –The times of charge and discharge in each measurement of touch channels. Range: 0 ~ 0xffff. Recommended typical value: Modify this value to make the measurement time around 1 ms.

**Returns**
- ESP_OK on success

`esp_err_t touch_pad_get_meas_time(uint16_t *sleep_cycle, uint16_t *meas_times)`

Get touch sensor times of charge and discharge and sleep time.

**Parameters**
- `sleep_cycle` – Pointer to accept sleep cycle number
- `meas_times` – Pointer to accept measurement times count.

**Returns**
- ESP_OK on success

`esp_err_t touch_pad_set_idle_channel_connect(touch_pad_conn_type_t type)`

Set the connection type of touch channels in idle status. When a channel is in measurement mode, other initialized channels are in idle mode. The touch channel is generally adjacent to the trace, so the connection state of the idle channel affects the stability and sensitivity of the test channel. The `CONN_HIGHZ` (high resistance) setting increases the sensitivity of touch channels. The `CONN_GND` (grounding) setting increases the stability of touch channels.

**Parameters** `type` –Select idle channel connect to high resistance state or ground.

**Returns**
- ESP_OK on success

`esp_err_t touch_pad_get_idle_channel_connect(touch_pad_conn_type_t *type)`

Get the connection type of touch channels in idle status. When a channel is in measurement mode, other initialized channels are in idle mode. The touch channel is generally adjacent to the trace, so the connection state of the idle channel affects the stability and sensitivity of the test channel. The `CONN_HIGHZ` (high resistance) setting increases the sensitivity of touch channels. The `CONN_GND` (grounding) setting increases the stability of touch channels.

**Parameters** `type` – Pointer to connection type.

**Returns**
- ESP_OK on success
**esp_err_t touch_pad_set_thresh** (*touch_pad_t* touch_num, *uint32_t* threshold)

Set the trigger threshold of touch sensor. The threshold determines the sensitivity of the touch sensor. The threshold is the original value of the trigger state minus the benchmark value.

**Note:** If set “TOUCH_PAD_THRESHOLD_MAX”, the touch is never be triggered.

**Parameters**
- *touch_num* – touch pad index
- *threshold* – threshold of touch sensor. Should be less than the max change value of touch.

**Returns**
- ESP_OK on success

**esp_err_t touch_pad_get_thresh** (*touch_pad_t* touch_num, *uint32_t* *threshold)

Get touch sensor trigger threshold.

**Parameters**
- *touch_num* – touch pad index
- *threshold* – pointer to accept threshold

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

**esp_err_t touch_pad_set_channel_mask** (*uint16_t* enable_mask)

Register touch channel into touch sensor scan group. The working mode of the touch sensor is cyclically scanned. This function will set the scan bits according to the given bitmask.

**Note:** If set this mask, the FSM timer should be stop firstly.

**Parameters**
- *enable_mask* – bitmask of touch sensorscangroup. e.g. TOUCH_PAD_NUM14 -> BIT(14)

**Returns**
- ESP_OK on success

**esp_err_t touch_pad_get_channel_mask** (*uint16_t* *enable_mask)

Get the touch sensor scan group bit mask.

**Parameters**
- *enable_mask* – Pointer to bitmask of touch sensorscangroup. e.g. TOUCH_PAD_NUM14 -> BIT(14)

**Returns**
- ESP_OK on success

**esp_err_t touch_pad_clear_channel_mask** (*uint16_t* enable_mask)

Clear touch channel from touch sensor scan group. The working mode of the touch sensor is cyclically scanned. This function will clear the scan bits according to the given bitmask.

**Note:** If clear all mask, the FSM timer should be stop firstly.

**Parameters**
- *enable_mask* – bitmask of touch sensorscangroup. e.g. TOUCH_PAD_NUM14 -> BIT(14)

**Returns**
Chapter 2. API Reference

- ESP_OK on success

```c
esp_err_t touch_pad_config(touch_pad_t touch_num)
```
Configure parameter for each touch channel.

**Note:** Touch num 0 is denoise channel, please use `touch_pad_denoise_enable` to set denoise function

**Parameters**
- `touch_num` – touch pad index

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG if argument wrong
- ESP_FAIL if touch pad not initialized

```c
esp_err_t touch_pad_reset(void)
```
Reset the FSM of touch module.

**Note:** Call this function after `touch_pad_fsm_stop`.

**Returns**
- ESP_OK Success

```c
touch_pad_t touch_pad_get_current_meas_channel(void)
```
Get the current measure channel.

**Note:** Should be called when touch sensor measurement is in cyclic scan mode.

**Returns**
- touch channel number

```c
uint32_t touch_pad_read_intr_status_mask(void)
```
Get the touch sensor interrupt status mask.

**Returns**
- touch interrupt bit

```c
esp_err_t touch_pad_intr_enable(touch_pad_intr_mask_t int_mask)
```
Enable touch sensor interrupt by bitmask.

**Note:** This API can be called in ISR handler.

**Parameters**
- `int_mask` – Pad mask to enable interrupts

**Returns**
- ESP_OK on success

```c
esp_err_t touch_pad_intr_disable(touch_pad_intr_mask_t int_mask)
```
Disable touch sensor interrupt by bitmask.

**Note:** This API can be called in ISR handler.

**Parameters**
- `int_mask` – Pad mask to disable interrupts
Chapter 2. API Reference

Returns
• ESP_OK on success

```c
esp_err_t touch_pad_intr_clear(touch_pad_intr_mask_t int_mask)
```
Clear touch sensor interrupt by bitmask.

**Parameters**
- `int_mask` – Pad mask to clear interrupts

**Returns**
• ESP_OK on success

```c
esp_err_t touch_pad_isr_register(intr_handler_t fn, void *arg, touch_pad_intr_mask_t intr_mask)
```
Register touch-pad ISR. The handler will be attached to the same CPU core that this function is running on.

**Parameters**
- `fn` – Pointer to ISR handler
- `arg` – Parameter for ISR
- `intr_mask` – Enable touch sensor interrupt handler by bitmask.

**Returns**
• ESP_OK Success
• ESP_ERR_INVALID_ARG Arguments error
• ESP_ERR_NO_MEM No memory

```c
esp_err_t touch_pad_timeout_set(bool enable, uint32_t threshold)
```
Enable/disable the timeout check and set timeout threshold for all touch sensor channels measurements. If enable: When the touch reading of a touch channel exceeds the measurement threshold, a timeout interrupt will be generated. If disable: the FSM does not check if the channel under measurement times out.

**Note:** The threshold compared with touch readings.

**Note:** In order to avoid abnormal short circuit of some touch channels. This function should be turned on. Ensure the normal operation of other touch channels.

**Parameters**
- `enable` – true(default): Enable the timeout check; false: Disable the timeout check.
- `threshold` – For all channels, the maximum value that will not be exceeded during normal operation.

**Returns**
• ESP_OK Success

```c
esp_err_t touch_pad_timeout_resume(void)
```
Call this interface after timeout to make the touch channel resume normal work. Point on the next channel to measure. If this API is not called, the touch FSM will stop the measurement after timeout interrupt.

**Note:** Call this API after finishes the exception handling by user.

**Returns**
• ESP_OK Success

```c
esp_err_t touch_pad_read_raw_data(touch_pad_t touch_num, uint32_t *raw_data)
```
get raw data of touch sensor.

**Note:** After the initialization is complete, the “raw_data” is max value. You need to wait for a measurement cycle before you can read the correct touch value.
Chapter 2. API Reference

Parameters
- `touch_num` - touch pad index
- `raw_data` - pointer to accept touch sensor value

Returns
- ESP_OK Success
- ESP_FAIL Touch channel 0 haven’t this parameter.

`esp_err_t touch_pad_read_benchmark(touch_pad_t touch_num, uint32_t *benchmark)`
get benchmark of touch sensor.

Note: After initialization, the benchmark value is the maximum during the first measurement period.

Parameters
- `touch_num` - touch pad index
- `benchmark` - pointer to accept touch sensor benchmark value

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch channel 0 haven’t this parameter.

`esp_err_t touch_pad_filter_read_smooth(touch_pad_t touch_num, uint32_t *smooth)`
Get smoothed data that obtained by filtering the raw data.

Parameters
- `touch_num` - touch pad index
- `smooth` - pointer to smoothed data

`esp_err_t touch_pad_reset_benchmark(touch_pad_t touch_num)`
Force reset benchmark to raw data of touch sensor.

Parameters `touch_num` - touch pad index

Returns
- ESP_OK Success

`esp_err_t touch_pad_filter_set_config(const touch_filter_config_t *filter_info)`
set parameter of touch sensor filter and detection algorithm. For more details on the detection algorithm, please refer to the application documentation.

Parameters `filter_info` - select filter type and threshold of detection algorithm

Returns
- ESP_OK Success

`esp_err_t touch_pad_filter_get_config(touch_filter_config_t *filter_info)`
get parameter of touch sensor filter and detection algorithm. For more details on the detection algorithm, please refer to the application documentation.

Parameters `filter_info` - select filter type and threshold of detection algorithm

Returns
- ESP_OK Success

`esp_err_t touch_pad_filter_enable(void)`
enable touch sensor filter for detection algorithm. For more details on the detection algorithm, please refer to the application documentation.

Returns
- ESP_OK Success

`esp_err_t touch_pad_filter_disable(void)`
disable touch sensor filter for detection algorithm. For more details on the detection algorithm, please refer to the application documentation.
Returns

• ESP_OK Success

```
est_err_t touch_pad_denoise_set_config (const touch_pad_denoise_t *denoise)
```

set parameter of denoise pad (TOUCH_PAD_NUM0). T0 is an internal channel that does not have a corresponding external GPIO. T0 will work simultaneously with the measured channel Tn. Finally, the actual measured value of Tn is the value after subtracting lower bits of T0. The noise reduction function filters out interference introduced simultaneously on all channels, such as noise introduced by power supplies and external EMI.

**Parameters**

- `denoise` - parameter of denoise

**Returns**

• ESP_OK Success

```
est_err_t touch_pad_denoise_get_config (touch_pad_denoise_t *denoise)
```

get parameter of denoise pad (TOUCH_PAD_NUM0).

**Parameters**

- `denoise` - Pointer to parameter of denoise

**Returns**

• ESP_OK Success

```
est_err_t touch_pad_denoise_enable (void)
```

enable denoise function. T0 is an internal channel that does not have a corresponding external GPIO. T0 will work simultaneously with the measured channel Tn. Finally, the actual measured value of Tn is the value after subtracting lower bits of T0. The noise reduction function filters out interference introduced simultaneously on all channels, such as noise introduced by power supplies and external EMI.

**Returns**

• ESP_OK Success

```
est_err_t touch_pad_denoise_disable (void)
```

disable denoise function.

**Returns**

• ESP_OK Success

```
est_err_t touch_pad_denoise_read_data (uint32_t *data)
```

Get denoise measure value (TOUCH_PAD_NUM0).

**Parameters**

- `data` - Pointer to receive denoise value

**Returns**

• ESP_OK Success

```
est_err_t touch_pad_waterproof_set_config (const touch_pad_waterproof_t *waterproof)
```

set parameter of waterproof function.

```
The waterproof function includes a shielded channel (TOUCH_PAD_NUM14) and a guard channel.
Guard pad is used to detect the large area of water covering the touch panel.
Shield pad is used to shield the influence of water droplets covering the touch panel.
It is generally designed as a grid and is placed around the touch buttons.
```

**Parameters**

- `waterproof` - parameter of waterproof

**Returns**

• ESP_OK Success

```
est_err_t touch_pad_waterproof_get_config (touch_pad_waterproof_t *waterproof)
```

get parameter of waterproof function.
**Parameters** waterproof – parameter of waterproof

**Returns**
- ESP_OK Success

`esp_err_t touch_pad_waterproof_enable (void)`

Enable parameter of waterproof function. Should be called after function `touch_pad_waterproof_set_config`.

**Returns**
- ESP_OK Success

`esp_err_t touch_pad_waterproof_disable (void)`

Disable parameter of waterproof function.

**Returns**
- ESP_OK Success

`esp_err_t touch_pad_proximity_enable (touch_pad_t touch_num, bool enabled)`

Enable/disable proximity function of touch channels. The proximity sensor measurement is the accumulation of touch channel measurements.

**Note:** Supports up to three touch channels configured as proximity sensors.

**Parameters**
- `touch_num` – touch pad index
- `enabled` – true: enable the proximity function; false: disable the proximity function

**Returns**
- ESP_OK: Configured correctly.
- ESP_ERR_INVALID_ARG: Touch channel number error.
- ESP_ERR_NOT_SUPPORTED: Don’t support configured.

`esp_err_t touch_pad_proximity_set_count (touch_pad_t touch_num, uint32_t count)`

Set measure count of proximity channel. The proximity sensor measurement is the accumulation of touch channel measurements.

**Note:** All proximity channels use the same `count` value. So please pass the parameter `TOUCH_PAD_MAX`.

**Parameters**
- `touch_num` – Touch pad index. In this version, pass the parameter `TOUCH_PAD_MAX`.
- `count` – The cumulative times of measurements for proximity pad. Range: 0 ~ 255.

**Returns**
- ESP_OK: Configured correctly.
- ESP_ERR_INVALID_ARG: Touch channel number error.

`esp_err_t touch_pad_proximity_get_count (touch_pad_t touch_num, uint32_t *count)`

Get measure count of proximity channel. The proximity sensor measurement is the accumulation of touch channel measurements.

**Note:** All proximity channels use the same `count` value. So please pass the parameter `TOUCH_PAD_MAX`.

**Parameters**
- `touch_num` – Touch pad index. In this version, pass the parameter `TOUCH_PAD_MAX`.
- `count` – The cumulative times of measurements for proximity pad. Range: 0 ~ 255.

**Returns**
- ESP_OK: Configured correctly.
• ESP_ERR_INVALID_ARG: Touch channel number error.

```c
enum esp_err_t touch_pad_proximity_get_data(touch_pad_t touch_num, uint32_t *measure_out)
```

Get the accumulated measurement of the proximity sensor. The proximity sensor measurement is the accumulation of touch channel measurements.

**Parameters**
- `touch_num` – touch pad index
- `measure_out` – If the accumulation process does not end, the `measure_out` is the process value.

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Touch num is not proximity

```c
enum esp_err_t touch_pad_sleep_channel_get_info(touch_pad_sleep_channel_t *slp_config)
```

Get parameter of touch sensor sleep channel. The touch sensor can works in sleep mode to wake up sleep.

**Note:** After the sleep channel is configured, Please use special functions for sleep channel. e.g. The user should use `touch_pad_sleep_channel_read_data` instead of `touch_pad_read_raw_data` to obtain the sleep channel reading.

**Parameters**
- `slp_config` – touch sleep pad config.

**Returns**
- ESP_OK Success

```c
enum esp_err_t touch_pad_sleep_channel_enable(touch_pad_t pad_num, bool enable)
```

Enable/Disable sleep channel function for touch sensor. The touch sensor can works in sleep mode to wake up sleep.

**Note:** ESP32S2 only support one sleep channel.

**Note:** After the sleep channel is configured, Please use special functions for sleep channel. e.g. The user should use `touch_pad_sleep_channel_read_data` instead of `touch_pad_read_raw_data` to obtain the sleep channel reading.

**Parameters**
- `pad_num` – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- `enable` – true: enable sleep pad for touch sensor; false: disable sleep pad for touch sensor;

**Returns**
- ESP_OK Success

```c
enum esp_err_t touch_pad_sleep_channel_enable_proximity(touch_pad_t pad_num, bool enable)
```

Enable/Disable proximity function for sleep channel. The touch sensor can works in sleep mode to wake up sleep.

**Note:** ESP32S2 only support one sleep channel.

**Parameters**
- `pad_num` – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
**enable** – true: enable proximity for sleep channel; false: disable proximity for sleep channel;

**Returns**
- ESP_OK Success

```c
esp_err_t touch_pad_sleep_set_threshold(touch_pad_t pad_num, uint32_t touch_thres)
```

Set the trigger threshold of touch sensor in deep sleep. The threshold determines the sensitivity of the touch sensor.

**Note:** In general, the touch threshold during sleep can use the threshold parameter parameters before sleep.

**Parameters**
- `pad_num` – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- `touch_thres` – touch sleep pad threshold

**Returns**
- ESP_OK Success

```c
esp_err_t touch_pad_sleep_get_threshold(touch_pad_t pad_num, uint32_t *touch_thres)
```

Get the trigger threshold of touch sensor in deep sleep. The threshold determines the sensitivity of the touch sensor.

**Note:** In general, the touch threshold during sleep can use the threshold parameter parameters before sleep.

**Parameters**
- `pad_num` – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- `touch_thres` – touch sleep pad threshold

**Returns**
- ESP_OK Success

```c
esp_err_t touch_pad_sleep_channel_read_benchmark(touch_pad_t pad_num, uint32_t *benchmark)
```

Read benchmark of touch sensor sleep channel.

**Parameters**
- `pad_num` – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- `benchmark` – pointer to accept touch sensor benchmark value

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG parameter is NULL

```c
esp_err_t touch_pad_sleep_channel_read_smooth(touch_pad_t pad_num, uint32_t *smooth_data)
```

Read smoothed data of touch sensor sleep channel. Smoothed data is filtered from the raw data.

**Parameters**
- `pad_num` – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- `smooth_data` – pointer to accept touch sensor smoothed data

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG parameter is NULL
### esp_err_t touch_pad_sleep_channel_read_data (touch_pad_t pad_num, uint32_t *raw_data)

Read raw data of touch sensor sleep channel.

**Parameters**
- **pad_num** – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- **raw_data** – Pointer to accept touch sensor raw data

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter is NULL

### esp_err_t touch_pad_sleep_channel_reset_benchmark (void)

Reset benchmark of touch sensor sleep channel.

**Returns**
- ESP_OK Success

### esp_err_t touch_pad_sleep_channel_read_proximity_cnt (touch_pad_t pad_num, uint32_t *proximity_cnt)

Read proximity count of touch sensor sleep channel.

**Parameters**
- **pad_num** – Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
- **proximity_cnt** – Pointer to accept touch sensor proximity count value

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter is NULL

### esp_err_t touch_pad_sleep_channel_set_work_time (uint16_t sleep_cycle, uint16_t meas_times)

Change the operating frequency of touch pad in deep sleep state. Reducing the operating frequency can effectively reduce power consumption. If this function is not called, the working frequency of touch in the deep sleep state is the same as that in the wake-up state.

**Parameters**
- **sleep_cycle** – The touch sensor will sleep after each measurement. sleep_cycle decide the interval between each measurement. $t_{sleep} = \frac{sleep\_cycle}{(RTC\_SLOW\_CLK\_frequency)}$. The approximate frequency value of RTC_SLOW_CLK can be obtained using rtc_clk_slow_freq_get_hz function.
- **meas_times** – The times of charge and discharge in each measure process of touch channels. The timer frequency is 8Mhz. Range: 0 - 0xffff. Recommended typical value: Modify this value to make the measurement time around 1ms.

**Returns**
- ESP_OK Success

### Header File

- components/driver/touch_sensor/include/driver/touch_sensor_common.h
- This header file can be included with:
  ```
  #include "driver/touch_sensor_common.h"
  ```
- This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:
  ```
  REQUIRES driver
  ```
  or
  ```
  PRIV_REQUIRES driver
  ```
Functions

```c
esp_err_t touch_pad_init (void)
Initailize touch module.
```

**Note:** If default parameter don’t match the usage scenario, it can be changed after this function.

**Returns**
- ESP_OK Success
- ESP_ERR_NO_MEM Touch pad init error
- ESP_ERR_NOT_SUPPORTED Touch pad is providing current to external XTAL

```c
esp_err_t touch_pad_deinit (void)
Un-install touch pad driver.
```

**Note:** After this function is called, other touch functions are prohibited from being called.

**Returns**
- ESP_OK Success
- ESP_FAIL Touch pad driver not initialized

```c
esp_err_t touch_pad_i0_init (touch_pad_t touch_num)
Initialize touch pad GPIO.
```

**Parameters**

- `touch_num` - touch pad index

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

```c
esp_err_t touch_pad_set_voltage (touch_high_volt_t refh, touch_low_volt_t refl, touch_volt_atten_t atten)
Set touch sensor high voltage threshold of charge. The touch sensor measures the channel capacitance value by charging and discharging the channel. So the high threshold should be less than the supply voltage.
```

**Parameters**

- `refh` - the value of DREFH
- `refl` - the value of DREFL
- `atten` - the attenuation on DREFH

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

```c
esp_err_t touch_pad_get_voltage (touch_high_volt_t *refh, touch_low_volt_t *refl, touch_volt_atten_t *atten)
Get touch sensor reference voltage..
```

**Parameters**

- `refh` - pointer to accept DREFH value
- `refl` - pointer to accept DREFL value
- `atten` - pointer to accept the attenuation on DREFH

**Returns**
- ESP_OK on success

```c
esp_err_t touch_pad_set_cnt_mode (touch_pad_t touch_num, touch_cnt_slope_t slope, touch_tie_opt_t opt)
Set touch sensor charge/discharge speed for each pad. If the slope is 0, the counter would always be zero. If the slope is 1, the charging and discharging would be slow, accordingly. If the slope is set 7, which is the maximum value, the charging and discharging would be fast.
```
Note: The higher the charge and discharge current, the greater the immunity of the touch channel, but it will increase the system power consumption.

Parameters
- `touch_num` – touch pad index
- `slope` – touch pad charge/discharge speed
- `opt` – the initial voltage

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

```c
esp_err_t touch_pad_get_cnt_mode(touch_pad_t touch_num, touch_cnt_slope_t *slope, touch_tie_opt_t *opt)
```

Get touch sensor charge/discharge speed for each pad.

Parameters
- `touch_num` – touch pad index
- `slope` – pointer to accept touch pad charge/discharge slope
- `opt` – pointer to accept the initial voltage

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

```c
esp_err_t touch_pad_isr_deregister(void (*fn)(void*), void* arg)
```

Deregister the handler previously registered using touch_pad_isr_handler_register.

Parameters
- `fn` – handler function to call (as passed to touch_pad_isr_handler_register)
- `arg` – argument of the handler (as passed to touch_pad_isr_handler_register)

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if a handler matching both fn and arg isn’t registered

```c
esp_err_t touch_pad_get_wakeup_status(touch_pad_t *pad_num)
```

Get the touch pad which caused wakeup from deep sleep.

Parameters `pad_num` – pointer to touch pad which caused wakeup

Returns
- ESP_OK Success
- ESP_ERR_INVALID_ARG parameter is NULL

```c
esp_err_t touch_pad_set_fsm_mode(touch_fsm_mode_t mode)
```

Set touch sensor FSM mode, the test action can be triggered by the timer, as well as by the software.

Parameters `mode` – FSM mode

Returns
- ESP_OK on success
- ESP_ERR_INVALID_ARG if argument is wrong

```c
esp_err_t touch_pad_get_fsm_mode(touch_fsm_mode_t *mode)
```

Get touch sensor FSM mode.

Parameters `mode` – pointer to accept FSM mode

Returns
- ESP_OK on success

```c
esp_err_t touch_pad_clear_status(void)
```

To clear the touch sensor channel active status.
Note: The FSM automatically updates the touch sensor status. It is generally not necessary to call this API to clear the status.

Returns
- ESP_OK on success

uint32_t touch_pad_get_status (void)
Get the touch sensor channel active status mask. The bit position represents the channel number. The 0/1 status of the bit represents the trigger status.

Returns
- The touch sensor status. e.g. Touch1 trigger status is status_mask & (BIT1).

bool touch_pad_meas_is_done (void)
Check touch sensor measurement status.

Returns
- True measurement is under way
- False measurement done

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
- components/soc/esp32s2/include/soc/touch_sensor_channel.h
- This header file can be included with:

```
#include "soc/touch_sensor_channel.h"
```

Macros

TOUCH_PAD_GPIO1_CHANNEL
TOUCH_PAD_NUM1_GPIO_NUM
TOUCH_PAD_GPIO2_CHANNEL
TOUCH_PAD_NUM2_GPIO_NUM
TOUCH_PAD_GPIO3_CHANNEL
TOUCH_PAD_NUM3_GPIO_NUM
TOUCH_PAD_GPIO4_CHANNEL
TOUCH_PAD_NUM4_GPIO_NUM
TOUCH_PAD_GPIO5_CHANNEL
Chapter 2. API Reference

TOUCH_PAD_NUM5_GPIO_NUM

TOUCH_PAD_GPIO6_CHANNEL

TOUCH_PAD_NUM6_GPIO_NUM

TOUCH_PAD_GPIO7_CHANNEL

TOUCH_PAD_NUM7_GPIO_NUM

TOUCH_PAD_GPIO8_CHANNEL

TOUCH_PAD_NUM8_GPIO_NUM

TOUCH_PAD_GPIO9_CHANNEL

TOUCH_PAD_NUM9_GPIO_NUM

TOUCH_PAD_GPIO10_CHANNEL

TOUCH_PAD_NUM10_GPIO_NUM

TOUCH_PAD_GPIO11_CHANNEL

TOUCH_PAD_NUM11_GPIO_NUM

TOUCH_PAD_GPIO12_CHANNEL

TOUCH_PAD_NUM12_GPIO_NUM

TOUCH_PAD_GPIO13_CHANNEL

TOUCH_PAD_NUM13_GPIO_NUM

TOUCH_PAD_GPIO14_CHANNEL

TOUCH_PAD_NUM14_GPIO_NUM

Header File

- components/hal/include/hal/touch_sensor_types.h
- This header file can be included with:

```c
#include "hal/touch_sensor_types.h"
```
Chapter 2. API Reference

Structures

struct `touch_pad_denoise`

Touch sensor denoise configuration

**Public Members**

`touch_pad_denoise_grade_t grade`

Select denoise range of denoise channel. Determined by measuring the noise amplitude of the denoise channel.

`touch_pad_denoise_cap_t cap_level`

Select internal reference capacitance of denoise channel. Ensure that the denoise readings are closest to the readings of the channel being measured. Use `touch_pad_denoise_read_data` to get the reading of denoise channel. The equivalent capacitance of the shielded channel can be calculated from the reading of denoise channel.

struct `touch_pad_waterproof`

Touch sensor waterproof configuration

**Public Members**

`touch_pad_t guard_ring_pad`

Waterproof. Select touch channel use for guard pad. Guard pad is used to detect the large area of water covering the touch panel.

`touch_pad_shield_driver_t shield_driver`

Waterproof. Shield channel drive capability configuration. Shield pad is used to shield the influence of water droplets covering the touch panel. When the waterproof function is enabled, Touch14 is set as shield channel by default. The larger the parasitic capacitance on the shielding channel, the higher the drive capability needs to be set. The equivalent capacitance of the shield channel can be estimated through the reading value of the denoise channel(Touch0).

struct `touch_filter_config`

Touch sensor filter configuration

**Public Members**

`touch_filter_mode_t mode`

Set filter mode. The input of the filter is the raw value of touch reading, and the output of the filter is involved in the judgment of the touch state.

`uint32_t debounce_cnt`

Set debounce count, such as $n$. If the measured values continue to exceed the threshold for $n+1$ times, the touch sensor state changes. Range: $0 \sim 7$

`uint32_t noise_thr`

Noise threshold coefficient. Higher = More noise resistance. The actual noise should be less than (noise coefficient * touch threshold). Range: $0 \sim 3$. The coefficient is $0: 4/8; 1: 3/8; 2: 2/8; 3: 1$;
uint32_t jitter_step
   Set jitter filter step size. Range: 0 ~ 15

touch_smooth_mode_t smh_lvl
   Level of filter applied on the original data against large noise interference.

struct touch_pad_sleep_channel_t
   Touch sensor channel sleep configuration

Public Members

touch_pad_t touch_num
   Set touch channel number for sleep pad. Only one touch sensor channel is supported in deep sleep mode.
   If clear the sleep channel, point this pad to TOUCH_PAD_NUM0

bool en_proximity
   enable proximity function for sleep pad

Macros

TOUCH_PAD_BIT_MASK_ALL

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_IDLE_CH_CONNECT_DEFAULT

TOUCH_PAD_THRESHOLD_MAX
   If set touch threshold max value, The touch sensor can’t be in touched status

TOUCH_PAD_SLEEP_CYCLE_DEFAULT
   Excessive total time will slow down the touch response. Too small measurement time will not be sampled enough, resulting in inaccurate measurements.

Note: The greater the duty cycle of the measurement time, the more system power is consumed. The number of sleep cycle in each measure process of touch channels. The timer frequency is RTC_SLOW_CLK (can be 150k or 32k depending on the options). Range: 0 ~ 0xffff
**TOUCH_PAD_MEASURE_CYCLE_DEFAULT**

The times of charge and discharge in each measure process of touch channels. The timer frequency is 8Mhz. Recommended typical value: Modify this value to make the measurement time around 1ms. Range: 0 ~ 0xffff

**TOUCH_PAD_INTR_MASK_ALL**

All touch interrupt type enable.

**TOUCH_PROXIMITY_MEAS_NUM_MAX**

Touch sensor proximity detection configuration

**TOUCH_DEBOUNCE_CNT_MAX**

**TOUCH_NOISE_THR_MAX**

**TOUCH_JITTER_STEP_MAX**

**Type Definitions**

typedef struct touch_pad_denoise touch_pad_denoise_t
    Touch sensor denoise configuration

typedef struct touch_pad_waterproof touch_pad_waterproof_t
    Touch sensor waterproof configuration

typedef struct touch_filter_config touch_filter_config_t
    Touch sensor filter configuration

**Enumerations**

enum touch_pad_t
    Touch pad channel

    Values:

    enumerator TOUCH_PAD_NUM0
        Touch pad channel 0 is GPIO4(ESP32)

    enumerator TOUCH_PAD_NUM1
        Touch pad channel 1 is GPIO0(ESP32) / GPIO1(ESP32-S2)

    enumerator TOUCH_PAD_NUM2
        Touch pad channel 2 is GPIO2(ESP32) / GPIO2(ESP32-S2)

    enumerator TOUCH_PAD_NUM3
        Touch pad channel 3 is GPIO15(ESP32) / GPIO3(ESP32-S2)

    enumerator TOUCH_PAD_NUM4
        Touch pad channel 4 is GPIO13(ESP32) / GPIO4(ESP32-S2)
enumerator TOUCH_PAD_NUM5
   Touch pad channel 5 is GPIO12(ESP32) / GPIO5(ESP32-S2)

enumerator TOUCH_PAD_NUM6
   Touch pad channel 6 is GPIO14(ESP32) / GPIO6(ESP32-S2)

enumerator TOUCH_PAD_NUM7
   Touch pad channel 7 is GPIO27(ESP32) / GPIO7(ESP32-S2)

enumerator TOUCH_PAD_NUM8
   Touch pad channel 8 is GPIO33(ESP32) / GPIO8(ESP32-S2)

enumerator TOUCH_PAD_NUM9
   Touch pad channel 9 is GPIO32(ESP32) / GPIO9(ESP32-S2)

enumerator TOUCH_PAD_NUM10
   Touch channel 10 is GPIO10(ESP32-S2)

enumerator TOUCH_PAD_NUM11
   Touch channel 11 is GPIO11(ESP32-S2)

enumerator TOUCH_PAD_NUM12
   Touch channel 12 is GPIO12(ESP32-S2)

enumerator TOUCH_PAD_NUM13
   Touch channel 13 is GPIO13(ESP32-S2)

enumerator TOUCH_PAD_NUM14
   Touch channel 14 is GPIO14(ESP32-S2)

enumerator TOUCH_PAD_MAX

enum touch_high_volt_t
   Touch sensor high reference voltage

   Values:

enumerator TOUCH_HVOLT_KEEP
   Touch sensor high reference voltage, no change

enumerator TOUCH_HVOLT_2V4
   Touch sensor high reference voltage, 2.4V

enumerator TOUCH_HVOLT_2V5
   Touch sensor high reference voltage, 2.5V

enumerator TOUCH_HVOLT_2V6
   Touch sensor high reference voltage, 2.6V
enumerator TOUCH_HVOLT_2V7
    Touch sensor high reference voltage, 2.7V

touch_low_volt_t
Values:
enumerator TOUCH_LVOLT_KEEP
    Touch sensor low reference voltage, no change

tenumerator TOUCH_LVOLT_0V5
    Touch sensor low reference voltage, 0.5V

tenumerator TOUCH_LVOLT_0V6
    Touch sensor low reference voltage, 0.6V

tenumerator TOUCH_LVOLT_0V7
    Touch sensor low reference voltage, 0.7V

tenumerator TOUCH_LVOLT_0V8
    Touch sensor low reference voltage, 0.8V

tenumerator TOUCH_LVOLT_MAX

enumerator TOUCH_HVOLT_ATTEN_1V5
    Touch sensor high reference voltage attenuation, 1.5V attenuation

tenumerator TOUCH_HVOLT_ATTEN_1V
    Touch sensor high reference voltage attenuation, 1.0V attenuation

tenumerator TOUCH_HVOLT_ATTEN_0V5
    Touch sensor high reference voltage attenuation, 0.5V attenuation

tenumerator TOUCH_HVOLT_ATTEN_0V
    Touch sensor high reference voltage attenuation, 0V attenuation

tenumerator TOUCH_HVOLT_ATTEN_MAX
enum `touch_cnt_slope_t`
   Touch sensor charge/discharge speed
   
   Values:
   
   - enumerator `TOUCH_PAD_SLOPE_0`
     Touch sensor charge/discharge speed, always zero
   - enumerator `TOUCH_PAD_SLOPE_1`
     Touch sensor charge/discharge speed, slowest
   - enumerator `TOUCH_PAD_SLOPE_2`
     Touch sensor charge/discharge speed
   - enumerator `TOUCH_PAD_SLOPE_3`
     Touch sensor charge/discharge speed
   - enumerator `TOUCH_PAD_SLOPE_4`
     Touch sensor charge/discharge speed
   - enumerator `TOUCH_PAD_SLOPE_5`
     Touch sensor charge/discharge speed
   - enumerator `TOUCH_PAD_SLOPE_6`
     Touch sensor charge/discharge speed
   - enumerator `TOUCH_PAD_SLOPE_7`
     Touch sensor charge/discharge speed, fast
   - enumerator `TOUCH_PAD_SLOPE_MAX`

enum `touch_tie_opt_t`
   Touch sensor initial charge level
   
   Values:
   
   - enumerator `TOUCH_PAD_TIE_OPT_LOW`
     Initial level of charging voltage, low level
   - enumerator `TOUCH_PAD_TIE_OPT_HIGH`
     Initial level of charging voltage, high level
   - enumerator `TOUCH_PAD_TIE_OPT_MAX`

enum `touch_fsm_mode_t`
   Touch sensor FSM mode
   
   Values:
   
   - enumerator `TOUCH_FSM_MODE_TIMER`
     To start touch FSM by timer
enumerator **TOUCH_FSM_MODE_SW**
   To start touch FSM by software trigger

enumerator **TOUCH_FSM_MODE_MAX**

enum **touch_trigger_mode_t**
   *Values:*

enumerator **TOUCH_TRIGGER_BELOW**
   Touch interrupt will happen if counter value is less than threshold.

enumerator **TOUCH_TRIGGER_ABOVE**
   Touch interrupt will happen if counter value is larger than threshold.

enumerator **TOUCH_TRIGGER_MAX**

enum **touch_trigger_src_t**
   *Values:*

enumerator **TOUCH_TRIGGER_SOURCE_BOTH**
   wakeup interrupt is generated if both SET1 and SET2 are “touched”

enumerator **TOUCH_TRIGGER_SOURCE_SET1**
   wakeup interrupt is generated if SET1 is “touched”

enumerator **TOUCH_TRIGGER_SOURCE_MAX**

enum **touch_pad_intr_mask_t**
   *Values:*

enumerator **TOUCH_PAD_INTR_MASK_DONE**
   Measurement done for one of the enabled channels.

enumerator **TOUCH_PAD_INTR_MASK_ACTIVE**
   Active for one of the enabled channels.

enumerator **TOUCH_PAD_INTR_MASK_INACTIVE**
   Inactive for one of the enabled channels.

enumerator **TOUCH_PADINTR MASK_SCAN DONE**
   Measurement done for all the enabled channels.

enumerator **TOUCH_PAD_INTR_MASK_TIMEOUT**
   Timeout for one of the enabled channels.

enum **touch_pad_denoise_grade_t**
   *Values:*

enumerator TOUCH_PAD_DENOISE_BIT12
    Denoise range is 12bit
enumerator TOUCH_PAD_DENOISE_BIT10
    Denoise range is 10bit
enumerator TOUCH_PAD_DENOISE_BIT8
    Denoise range is 8bit
enumerator TOUCH_PAD_DENOISE_BIT4
    Denoise range is 4bit
enumerator TOUCH_PAD_DENOISE_MAX

enum touch_pad_denoise_cap_t
    Values:
        enumerator TOUCH_PAD_DENOISE_CAP_L0
            Denoise channel internal reference capacitance is 5pf
        enumerator TOUCH_PAD_DENOISE_CAP_L1
            Denoise channel internal reference capacitance is 6.4pf
        enumerator TOUCH_PAD_DENOISE_CAP_L2
            Denoise channel internal reference capacitance is 7.8pf
        enumerator TOUCH_PAD_DENOISE_CAP_L3
            Denoise channel internal reference capacitance is 9.2pf
        enumerator TOUCH_PAD_DENOISE_CAP_L4
            Denoise channel internal reference capacitance is 10.6pf
        enumerator TOUCH_PAD_DENOISE_CAP_L5
            Denoise channel internal reference capacitance is 12.0pf
        enumerator TOUCH_PAD_DENOISE_CAP_L6
            Denoise channel internal reference capacitance is 13.4pf
        enumerator TOUCH_PAD_DENOISE_CAP_L7
            Denoise channel internal reference capacitance is 14.8pf
        enumerator TOUCH_PAD_DENOISE_CAP_MAX

enum touch_pad_shield_driver_t
    Touch sensor shield channel drive capability level
    Values:
enumerator **TOUCH_PAD_SHIELD_DRV_L0**  
The max equivalent capacitance in shield channel is 40pf

enumerator **TOUCH_PAD_SHIELD_DRV_L1**  
The max equivalent capacitance in shield channel is 80pf

enumerator **TOUCH_PAD_SHIELD_DRV_L2**  
The max equivalent capacitance in shield channel is 120pf

enumerator **TOUCH_PAD_SHIELD_DRV_L3**  
The max equivalent capacitance in shield channel is 160pf

enumerator **TOUCH_PAD_SHIELD_DRV_L4**  
The max equivalent capacitance in shield channel is 200pf

enumerator **TOUCH_PAD_SHIELD_DRV_L5**  
The max equivalent capacitance in shield channel is 240pf

enumerator **TOUCH_PAD_SHIELD_DRV_L6**  
The max equivalent capacitance in shield channel is 280pf

enumerator **TOUCH_PAD_SHIELD_DRV_L7**  
The max equivalent capacitance in shield channel is 320pf

enumerator **TOUCH_PAD_SHIELD_DRV_MAX**

enum **touch_pad_conn_type_t**  
Touch channel idle state configuration  
**Values:**

enumerator **TOUCH_PAD_CONN_HIGHZ**  
Idle status of touch channel is high resistance state

enumerator **TOUCH_PAD_CONN_GND**  
Idle status of touch channel is ground connection

enumerator **TOUCH_PAD_CONN_MAX**

enum **touch_filter_mode_t**  
Touch channel IIR filter coefficient configuration.

---

**Note:** On ESP32S2. There is an error in the IIR calculation. The magnitude of the error is twice the filter coefficient. So please select a smaller filter coefficient on the basis of meeting the filtering requirements. Recommended filter coefficient selection **IIR_16**.

**Values:**
enumerator **TOUCH_PAD_FILTER_IIR_4**
    The filter mode is first-order IIR filter. The coefficient is 4.

tenumerator **TOUCH_PAD_FILTER_IIR_8**
    The filter mode is first-order IIR filter. The coefficient is 8.

tenumerator **TOUCH_PAD_FILTER_IIR_16**
    The filter mode is first-order IIR filter. The coefficient is 16 (Typical value).

tenumerator **TOUCH_PAD_FILTER_IIR_32**
    The filter mode is first-order IIR filter. The coefficient is 32.

tenumerator **TOUCH_PAD_FILTER_IIR_64**
    The filter mode is first-order IIR filter. The coefficient is 64.

tenumerator **TOUCH_PAD_FILTER_IIR_128**
    The filter mode is first-order IIR filter. The coefficient is 128.

tenumerator **TOUCH_PAD_FILTER_IIR_256**
    The filter mode is first-order IIR filter. The coefficient is 256.

tenumerator **TOUCH_PAD_FILTER_JITTER**
    The filter mode is jitter filter

enumerator **TOUCH_PAD_FILTER_MAX**

**enum touch_smooth_mode_t**

Level of filter applied on the original data against large noise interference.

---

**Note:** On ESP32S2. There is an error in the IIR calculation. The magnitude of the error is twice the filter coefficient. So please select a smaller filter coefficient on the basis of meeting the filtering requirements. Recommended filter coefficient selection **IIR_2**.

---

**Values:**

enumerator **TOUCH_PAD_SMOOTH_OFF**
    No filtering of raw data.

tenumerator **TOUCH_PAD_SMOOTH_IIR_2**
    Filter the raw data. The coefficient is 2 (Typical value).

tenumerator **TOUCH_PAD_SMOOTH_IIR_4**
    Filter the raw data. The coefficient is 4.

tenumerator **TOUCH_PAD_SMOOTH_IIR_8**
    Filter the raw data. The coefficient is 8.

tenumerator **TOUCH_PAD_SMOOTH_MAX**
2.5.25 Touch Element

Overview

The Touch Element Library is a highly abstracted element library designed on the basis of the touch sensor driver. The library provides a unified and user-friendly software interface to quickly build capacitive touch sensor applications. For more information about the touch sensor driver API, see Touch Sensor.

Architecture

The Touch Element library configures touch sensor peripherals via the touch sensor driver. However, some necessary hardware parameters should be passed to `touch_element_install()` and will be configured automatically only after calling `touch_element_start()`. This sequential order is essential because configuring these parameters has a significant impact on the run-time system. Therefore, they must be configured after calling the start function to ensure the system functions properly.

These parameters include touch channel threshold, driver-level of waterproof shield sensor, etc. The Touch Element library sets the touch sensor interrupt and the esp_timer routine up, and the hardware information of the touch sensor (channel state, channel number) will be obtained in the touch sensor interrupt service routine. When the specified channel event occurs, the hardware information is passed to the esp_timer callback routine, which then dispatches the touch sensor channel information to the touch elements (such as button, slider, etc.). The library then runs a specified algorithm to update the touch element’s state or calculate its position and dispatches the result accordingly.

So when using the Touch Element library, you are relieved from the implementation details of the touch sensor peripheral. The library handles most of the hardware information and passes the more meaningful messages to the event handler routine.

The workflow of the Touch Element library is illustrated in the picture below.

The features in relation to the Touch Element library in ESP32-S2 are given in the table below.

<table>
<thead>
<tr>
<th>Features</th>
<th>ESP32S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch Element waterproof</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Touch Element button</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Touch Element slider</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Touch Element matrix button</td>
<td>✓✓✓</td>
</tr>
</tbody>
</table>

Peripheral

ESP32-S2 integrates one touch sensor peripheral with several physical channels.

- 14 physical capacitive touch channels
- Timer or software FSM trigger mode
- Up to 5 kinds of interrupt (Upper threshold and lower threshold interrupt, measure one channel finish and measure all channels finish interrupt, measurement timeout interrupt)
- Sleep mode wakeup source
- Hardware internal de-noise
- Hardware filter
- Hardware waterproof sensor
- Hardware proximity sensor

The channels are located as follows:
Fig. 22: Touch Element architecture
### Channel

<table>
<thead>
<tr>
<th>Channel</th>
<th>ESP32-S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 0</td>
<td>GPIO 0 (reserved)</td>
</tr>
<tr>
<td>Channel 1</td>
<td>GPIO 1</td>
</tr>
<tr>
<td>Channel 2</td>
<td>GPIO 2</td>
</tr>
<tr>
<td>Channel 3</td>
<td>GPIO 3</td>
</tr>
<tr>
<td>Channel 4</td>
<td>GPIO 4</td>
</tr>
<tr>
<td>Channel 5</td>
<td>GPIO 5</td>
</tr>
<tr>
<td>Channel 6</td>
<td>GPIO 6</td>
</tr>
<tr>
<td>Channel 7</td>
<td>GPIO 7</td>
</tr>
<tr>
<td>Channel 8</td>
<td>GPIO 8</td>
</tr>
<tr>
<td>Channel 9</td>
<td>GPIO 9</td>
</tr>
<tr>
<td>Channel 10</td>
<td>GPIO 10</td>
</tr>
<tr>
<td>Channel 11</td>
<td>GPIO 11</td>
</tr>
<tr>
<td>Channel 12</td>
<td>GPIO 12</td>
</tr>
<tr>
<td>Channel 13</td>
<td>GPIO 13</td>
</tr>
<tr>
<td>Channel 14</td>
<td>GPIO 14</td>
</tr>
</tbody>
</table>

The channels are located as follows:

<table>
<thead>
<tr>
<th>Channel</th>
<th>ESP32-S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 0</td>
<td>GPIO 0 (effective)</td>
</tr>
<tr>
<td>Channel 1</td>
<td>GPIO 1</td>
</tr>
<tr>
<td>Channel 2</td>
<td>GPIO 2</td>
</tr>
<tr>
<td>Channel 3</td>
<td>GPIO 3</td>
</tr>
<tr>
<td>Channel 4</td>
<td>GPIO 4</td>
</tr>
<tr>
<td>Channel 5</td>
<td>GPIO 5</td>
</tr>
<tr>
<td>Channel 6</td>
<td>GPIO 6</td>
</tr>
<tr>
<td>Channel 7</td>
<td>GPIO 7</td>
</tr>
<tr>
<td>Channel 8</td>
<td>GPIO 8</td>
</tr>
<tr>
<td>Channel 9</td>
<td>GPIO 9</td>
</tr>
</tbody>
</table>

### Terminology

The terms used in relation to the Touch Element library are given below.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch sensor</td>
<td>Touch sensor peripheral inside the chip</td>
</tr>
<tr>
<td>Touch channel</td>
<td>Touch sensor channels inside the touch sensor peripheral</td>
</tr>
<tr>
<td>Touch pad</td>
<td>Off-chip physical solder pad, generally inside the PCB</td>
</tr>
<tr>
<td>De-noise channel</td>
<td>Internal de-noise channel, which is always Channel 0 and is reserved</td>
</tr>
<tr>
<td>Shield sensor</td>
<td>One of the waterproof sensors for detecting droplets in small areas and compensating for the influence of water drops on measurements</td>
</tr>
<tr>
<td>Guard sensor</td>
<td>One of the waterproof sensors for detecting extensive wading and to temporarily disable the touch sensor</td>
</tr>
<tr>
<td>Shield channel</td>
<td>The channel that waterproof shield sensor connected to, which is always Channel 14</td>
</tr>
<tr>
<td>Guard channel</td>
<td>The channel that waterproof guard sensor connected to</td>
</tr>
<tr>
<td>Shield pad</td>
<td>Off-chip physical solder pad, generally is grids, and is connected to shield the sensor</td>
</tr>
<tr>
<td>Guard pad</td>
<td>Off-chip physical solder pad, usually a ring, and is connected to the guard sensor</td>
</tr>
</tbody>
</table>
Fig. 23: Touch sensor application system components
**Touch Sensor Signal**  Each touch sensor is able to provide the following types of signals:

- **Raw**: The Raw signal is the unfiltered signal from the touch sensor.
- **Smooth**: The Smooth signal is a filtered version of the Raw signal via an internal hardware filter.
- **Benchmark**: The Benchmark signal is also a filtered signal that filters out extremely low-frequency noise.

All of these signals can be obtained using touch sensor driver API.

![Touch Sensor Signal](image)

**Fig. 24: Touch sensor signals**

**Touch Sensor Signal Threshold**  The Touch Sensor Threshold value is a configurable threshold value used to determine when a touch sensor is touched or not. When the difference between the Smooth signal and the Benchmark signal becomes greater than the threshold value (i.e., \((\text{smooth} - \text{benchmark}) > \text{threshold}\)), the touch channel’s state will be changed and a touch interrupt will be triggered simultaneously.

![Touch Sensor Signal Threshold](image)

**Fig. 25: Touch sensor signal threshold**

**Sensitivity**  Important performance parameter of the touch sensor, the larger it is, the better touch the sensor performs. It could be calculated by the format below:

\[
\text{Sensitivity} = \frac{\text{Signal}_\text{press} - \text{Signal}_\text{release}}{\text{Signal}_\text{release}} = \frac{\text{Signal}_\text{delta}}{\text{Signal}_\text{benchmark}}
\]

**Waterproof**  Waterproof is the hardware feature of a touch sensor which has a guard sensor and shield sensor (always connect to Channel 14) that has the ability to resist a degree of influence of water drop and detect the water stream.

**Touch Button**  The touch button consumes one channel of the touch sensor, and it looks like as the picture below:
Fig. 26: Touch button
**Touch Slider**  The touch slider consumes several channels (at least three channels) of the touch sensor, the more channels consumed, the higher resolution and accuracy position it performs. The touch slider looks like as the picture below:

![Touch Slider Diagram](image)

**Touch Matrix**  The touch matrix button consumes several channels (at least $2 + 2 = 4$ channels), and it gives a solution to use fewer channels and get more buttons. ESP32-S2 supports up to 49 buttons. The touch matrix button looks like as the picture below:

**Touch Element Library Usage**

Using this library should follow the initialization flow below:

1. To initialize the Touch Element library by calling `touch_element_install()`.  
2. To initialize touch elements (button/slider etc) by calling `touch_xxxx_install()`.  
3. To create a new element instance by calling `touch_xxxx_create()`.  
4. To subscribe events by calling `touch_xxxx_subscribe_event()`.  
5. To choose a dispatch method by calling `touch_xxxx_set_dispatch_method()` that tells the library how to notify you while the subscribed event occurs.
Fig. 28: Touch matrix
6. If dispatch by callback, call `touch_xxxx_set_callback()` to set the event handler function.
7. To start the Touch Element library by calling `touch_element_start()`.
8. If dispatch by callback, the callback will be called by the driver core when an event happens, no need to do anything; If dispatch by event task, create an event task and call `touch_element_message_receive()` to obtain messages in a loop.
9. (Optional) If you want to suspend the Touch Element run-time system or for some reason that could not obtain the touch element message, `touch_element_stop()` should be called to suspend the Touch Element system and then resume it by calling `touch_element_start()` again.

In code, the flow above may look like as follows:

```c
static touch_xxx_handle_t element_handle; //Declare a touch element handle

//Define the subscribed event handler
void event_handler(touch_xxx_handle_t out_handle, touch_xxx_message_t out_message, __void *arg)
{
    //Event handler logic
}

void app_main()
{
    //Using the default initializer to config Touch Element library
    touch_elem_global_config_t global_config = TOUCH_ELEM_GLOBAL_DEFAULT_CONFIG();
    touch_element_install(&global_config);

    //Using the default initializer to config Touch elements
    touch_xxx_global_config_t elem_global_config = TOUCH_XXXX_GLOBAL_DEFAULT_CONFIG();
    touch_xxx_install(&elem_global_config);

    //Create a new instance
    touch_xxx_config_t element_config = {
        ...
        ...
    };
    touch_xxx_create(&element_config, &element_handle);

    //Subscribe the specified events by using the event mask
    touch_xxx_subscribe_event(element_handle, TOUCH_ELEM_EVENT_ON_PRESS | TOUCH_ELEM_EVENT_ON_RELEASE, NULL);

    //Choose CALLBACK as the dispatch method
    touch_xxx_set_dispatch_method(element_handle, TOUCH_ELEM_DISP_CALLBACK);

    //Register the callback routine
    touch_xxx_set_callback(element_handle, event_handler);

    //Start Touch Element library processing
    touch_element_start();
}
```

**Initialization**

1. To initialize the Touch Element library, you have to configure the touch sensor peripheral and Touch Element library by calling `touch_element_install()` with `touch_elem_global_config_t`, the default initializer is available in `TOUCH_ELEM_GLOBAL_DEFAULT_CONFIG()` and this default configuration is suitable for the most general application scene, and it is suggested not to change the default configuration before fully understanding Touch Sensor peripheral because some changes might bring several impacts to the system.
2. To initialize the specified element, all the elements will not work before its constructor `touch_xxxx_install()` is called so as to save memory, so you have to call the constructor of
each used touch element respectively, to set up the specified element.

**Touch Element Instance Startup**

1. To create a new touch element instance, call `touch_xxxx_create()`, select a channel, and provide its Sensitivity value for the new element instance.
2. To subscribe to events, call `touch_xxxx_subscribe_event()`. The Touch Element library offers several events, and the event mask is available in components/touch_element/include/touch_element/touch_element.h. You can use these event masks to subscribe to specific events individually or combine them to subscribe to multiple events.
3. To configure the dispatch method, use `touch_xxxx_set_dispatch_method()`. The Touch Element library provides two dispatch methods: TOUCH_ELEM_DISP_EVENT and TOUCH_ELEM_DISP_CALLBACK. These methods allow you to obtain the touch element message and handle it using different approaches.

**Events Processing**  If TOUCH_ELEM_DISP_EVENT dispatch method is configured, you need to start up an event handler task to obtain the touch element message, all the elements’ raw message could be obtained by calling `touch_element_message_receive()`, then extract the element-class-specific message by calling the corresponding message decoder with `touch_xxxx_get_message()` to get the touch element’s extracted message; If TOUCH_ELEM_DISP_CALLBACK dispatch method is configured, you need to pass an event handler by calling `touch_xxxx_set_callback()` before the touch element starts working, all the element’s extracted message will be passed to the event handler function.

**Warning:** Since the event handler function runs on the core of the element library, i.e., in the esp_timer callback routine, please avoid performing operations that may cause blocking or delays, such as calling vTaskDelay().

In code, the events handle procedure may look like as follows:

```c
/* ---------------------------------------------- TOUCH_ELEM_DISP_EVENT -------- */
___------------------------------- */
void element_handler_task(void *arg)
{
    touch_elem_message_t element_message;
    while(1) {
        if (touch_element_message_receive(&element_message, Timeout) == ESP_OK) {
            const touch_xxxx_message_t *extracted_message = touch_xxxx_getメッセージ(&element_message); //Decode message
            ... //Event handler logic
        }
    }
}
void app_main()
{
    ...
    touch_xxxx_set_dispatch_method(element_handle, TOUCH_ELEM_DISP_EVENT); //Set
    __TOUCH_ELEM_DISP_EVENT as the dispatch method
    xTaskCreate(element_handler_task, "element_handler_task", 2048, NULL, 5, __
    __NULL); //Create a handler task
    ...
    /* ---------------------------------------------- TOUCH_ELEM_DISP_CALLBACK -------- */
    ```
void element_handler(touch_xxxx_handle_t out_handle, touch_xxxx_message_t out_message, void *arg)
{
    //Event handler logic
}

void app_main()
{
    ...
    touch_xxxx_set_dispatch_method(element_handle, TOUCH_ELEM_DISP_CALLBACK); // Set TOUCH_ELEM_DISP_CALLBACK as the dispatch method
    touch_xxxx_set_callback(element_handle, element_handler); //Register an event handler function
    ...
} /* ----------------------------------------------- */

Waterproof Usage

1. The waterproof shield sensor is always-on after Touch Element waterproof is initialized, however, the waterproof guard sensor is optional, hence if the you do not need the guard sensor, TOUCH_WATERPROOF_GUARD_NOUSE has to be passed to touch_element_waterproof_install() by the configuration struct.

2. To associate the touch element with the guard sensor, pass the touch element’s handle to the Touch Element waterproof’s masked list by calling touch_element_waterproof_add(). By associating a touch element with the Guard sensor, the touch element will be disabled when the guard sensor is triggered by a stream of water so as to protect the touch element.

The Touch Element Waterproof example is available in peripherals/touch_sensor/touch_element/touch_element_waterproof directory.

In code, the waterproof configuration may look as follows:

void app_main()
{
    ...
    touch_xxxx_install(); //Initialize instance (button, slider, etc)
    touch_xxxx_create(element_handle); //Create a new Touch element
    ...
    touch_element_waterproof_install(); //Initialize Touch Element waterproof
    touch_element_waterproof_add(element_handle); //Let an element associate with the guard sensor
    ...
}

Wakeup from Light/Deep-sleep Mode   Only Touch Button can be configured as a wake-up source.

Light- or Deep-sleep modes are both supported to be wakened up by a touch sensor. For the Light-sleep mode, any installed touch button can wake it up. But only the sleep button can wake up from Deep-sleep mode, and the touch sensor will do a calibration immediately, the reference value will be calibrated to a wrong value if our finger does not remove timely. Though the wrong reference value recovers after the finger removes away and has no effect on the
driver logic, if you do not want to see a wrong reference value while waking up from Deep-sleep mode, you can call `touch_element_sleep_enable_wakeup_calibration()` to disable the wakeup calibration.

The Touch Element Wakeup example is available in `system/light_sleep` directory.

```c
void app_main()
{
    ...
    touch_element_install();
    touch_button_install();       //Initialize the touch button
    touch_button_create(&element_handle);  //Create a new Touch element
    ...

    // ESP_ERROR_CHECK(touch_element_enable_light_sleep(&sleep_config));
    ESP_ERROR_CHECK(touch_element_enable_deep_sleep(button_handle[0], &sleep_config));
    // ESP_ERROR_CHECK(touch_element_sleep_enable_wakeup_calibration(button_handle[0], false)); // (optional) Disable wakeup calibration to prevent updating the baseline to a wrong value
    touch_element_start();
    ...
}
```

Application Example

All the Touch Element library examples could be found in the `peripherals/touch_sensor/touch_element` directory of ESP-IDF examples.

API Reference - Touch Element Core

Header File

- components/touch_element/include/touch_element/touch_element.h

This header file can be included with:

```c
#include "touch_element/touch_element.h"
```

This header file is a part of the API provided by the `touch_element` component. To declare that your component depends on `touch_element`, add the following to your CMakeLists.txt:

```cmake
REQUIRES touch_element
```

or

```cmake
PRIV_REQUIRES touch_element
```

Functions

`esp_err_t touch_element_install (const touch_elem_global_config_t *global_config)`

Touch element processing initialization.

**Note:** To reinitialize the touch element object, call `touch_element_uninstall()` first

**Parameters**

- `global_config` [in] Global initialization configuration structure

**Returns**

- ESP_OK: Successfully initialized
Chapter 2. API Reference

- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_INVALID_STATE: Touch element is already initialized
- Others: Unknown touch driver layer or lower layer error

```c
esp_err_t touch_element_start (void)
```

Touch element processing start.

This function starts the touch element processing system

**Note:** This function must only be called after all the touch element instances finished creating

**Returns**

- ESP_OK: Successfully started to process
- Others: Unknown touch driver layer or lower layer error

```c
esp_err_t touch_element_stop (void)
```

Touch element processing stop.

This function stops the touch element processing system

**Note:** This function must be called before changing the system (hardware, software) parameters

**Returns**

- ESP_OK: Successfully stopped to process
- Others: Unknown touch driver layer or lower layer error

```c
void touch_element_uninstall (void)
```

Release resources allocated using touch_element_install.

```c
ext esp_err_t touch_element_message_receive (touch_elem_message_t *element_message, uint32_t ticks_to_wait)
```

Get current event message of touch element instance.

This function will receive the touch element message (handle, event type, etc) from te_event_give(). It will block until a touch element event or a timeout occurs.

**Parameters**

- `element_message` - [out] Touch element event message structure
- `ticks_to_wait` - [in] Number of FreeRTOS ticks to block for waiting event

**Returns**

- ESP_OK: Successfully received touch element event
- ESP_ERR_INVALID_STATE: Touch element library is not initialized
- ESP_ERR_INVALID_ARG: element_message is null
- ESP_ERR_TIMEOUT: Timed out waiting for event

```c
esp_err_t touch_element_waterproof_install (const touch_elem_waterproof_config_t *waterproof_config)
```

Touch element waterproof initialization.

This function enables the hardware waterproof, then touch element system uses Shield-Sensor and Guard-Sensor to mitigate the influence of water-drop and water-stream.

**Note:** If the waterproof function is used, Shield-Sensor can not be disabled and it will use channel 14 as its internal channel. Hence, the user can not use channel 14 for another propose. And the Guard-Sensor is not necessary since it is optional.
Chapter 2. API Reference

**Note:** Shield-Sensor: It always uses channel 14 as the shield channel, so user must connect the channel 14 and Shield-Layer in PCB since it will generate a synchronous signal automatically.

**Note:** Guard-Sensor: This function is optional. If used, the user must connect the guard channel and Guard-Ring in PCB. Any channels user wants to protect should be added into Guard-Ring in PCB.

### Parameters

**waterproof_config** – [in] Waterproof configuration

**Returns**

- ESP_OK: Successfully initialized
- ESP_ERR_INVALID_STATE: Touch element library is not initialized
- ESP_ERR_INVALID_ARG: waterproof_config is null or invalid Guard-Sensor channel
- ESP_ERR_NO_MEM: Insufficient memory

```c
void touch_element_waterproof_uninstall (void)
Release resources allocated using touch_element_waterproof_install()
```

```c
esp_err_t touch_element_waterproof_add (touch_elem_handle_t element_handle)
Add a masked handle to protect while Guard-Sensor has been triggered.
This function will add an application handle (button, slider, etc) as a masked handle. While Guard-Sensor
has been triggered, waterproof function will start working and lock the application internal state. While the
influence of water is reduced, the application will be unlock and reset into IDLE state.

**Note:** The waterproof protection logic must follow the real circuit in PCB, it means that all of the channels
inside the input handle must be inside the Guard-Ring in real circuit.

### Parameters

**element_handle** – [in] Touch element instance handle

**Returns**

- ESP_OK: Successfully added a masked handle
- ESP_ERR_INVALID_STATE: Waterproof is not initialized
- ESP_ERR_INVALID_ARG: element_handle is null

```c
esp_err_t touch_element_waterproof_remove (touch_elem_handle_t element_handle)
Remove a masked handle to protect.
This function will remove an application handle from masked handle table.
```

### Parameters

**element_handle** – [in] Touch element instance handle

**Returns**

- ESP_OK: Successfully removed a masked handle
- ESP_ERR_INVALID_STATE: Waterproof is not initialized
- ESP_ERR_INVALID_ARG: element_handle is null
- ESP_ERR_NOT_FOUND: Failed to search element_handle from waterproof mask_handle list

```c
esp_err_t touch_element_enable_light_sleep (const touch_elem_sleep_config_t *sleep_config)
Touch element light sleep initialization.

**Note:** It should be called after touch button element installed. Any of installed touch element can wake up
from the light sleep

### Parameters

**sleep_config** – [in] Sleep configurations, set NULL to use default config

**Returns**

```c
```
• ESP_OK: Successfully initialized touch sleep
• ESP_ERR_INVALID_STATE: Touch element is not installed or touch sleep has been installed
• ESP_ERR_INVALID_ARG: inputed argument is NULL
• ESP_ERR_NO_MEM: no memory for touch sleep struct
• ESP_ERR_NOT_SUPPORTED: inputed wakeup_elem_handle is not touch_button_handle_t type, currently only touch_button_handle_t supported

`esp_err_t touch_element_disable_light_sleep (void)`

Release the resources that allocated by touch_element_enable_deep_sleep()

This function will also disable the touch sensor to wake up the device

Returns
• ESP_OK: uninstall success
• ESP_ERR_INVALID_STATE: touch sleep has not been installed

`esp_err_t touch_element_enable_deep_sleep (touch_elem_handle_t wakeup_elem_handle, const touch_elem_sleep_config_t *sleep_config)`

Touch element deep sleep initialization.

This function will enable the device wake-up from deep sleep or light sleep by touch sensor

**Note:** It should be called after touch button element installed. Only one touch button can be registered as the deep sleep wake-up button

**Parameters**
• `wakeup_elem_handle` [in] Touch element instance handle for waking up the device, only support button element
• `sleep_config` [in] Sleep configurations, set NULL to use default config

Returns
• ESP_OK: Successfully initialized touch sleep
• ESP_ERR_INVALID_STATE: Touch element is not installed or touch sleep has been installed
• ESP_ERR_INVALID_ARG: inputed argument is NULL
• ESP_ERR_NO_MEM: no memory for touch sleep struct
• ESP_ERR_NOT_SUPPORTED: inputed wakeup_elem_handle is not touch_button_handle_t type, currently only touch_button_handle_t supported

`esp_err_t touch_element_disable_deep_sleep (void)`

Release the resources that allocated by touch_element_enable_deep_sleep()

This function will also disable the touch sensor to wake up the device

Returns
• ESP_OK: uninstall success
• ESP_ERR_INVALID_STATE: touch sleep has not been installed

`esp_err_t touch_element_sleep_enable_wakeup_calibration (touch_elem_handle_t element_handle, bool en)`

Touch element wake up calibrations.

This function will also disable the touch sensor to wake up the device

Returns
• ESP_OK: uninstall success
• ESP_ERR_INVALID_STATE: touch sleep has not been installed
Structures

struct `touch_elem_sw_config_t`

Touch element software configuration.

Public Members

float `waterproof_threshold_divider`

Waterproof guard channel threshold divider.

uint8_t `processing_period`

Processing period (ms)

uint8_t `intr_message_size`

Interrupt message queue size.

uint8_t `event_message_size`

Event message queue size.

struct `touch_elem_hw_config_t`

Touch element hardware configuration.

Public Members

touch_high_volt_t `upper_voltage`

Touch sensor channel upper charge voltage.

touch_volt_atten_t `voltage_attenuation`

Touch sensor channel upper charge voltage attenuation (Diff voltage is upper - attenuation - lower)

touch_low_volt_t `lower_voltage`

Touch sensor channel lower charge voltage.

touch_pad_conn_type_t `suspend_channel_polarity`

Suspend channel polarity (High Impedance State or GND)

touch_pad_denoise_grade_t `denoise_level`

Internal de-noise level.

touch_pad_denoise_cap_t `denoise_equivalent_cap`

Internal de-noise channel (Touch channel 0) equivalent capacitance.

touch_smooth_mode_t `smooth_filter_mode`

Smooth value filter mode (This only apply to touch_pad_filter_read_smooth())

touch_filter_mode_t `benchmark_filter_mode`

Benchmark filter mode.


**uint16_t sample_count**

The count of sample in each measurement of touch sensor.

**uint16_t sleep_cycle**

The cycle (RTC slow clock) of sleep.

**uint8_t benchmark_debounce_count**

Benchmark debounce count.

**uint8_t benchmark_calibration_threshold**

Benchmark calibration threshold.

**uint8_t benchmark_jitter_step**

Benchmark jitter filter step (This only works at while benchmark filter mode is jitter filter)

**struct touch_elem_global_config_t**

Touch element global configuration passed to touch_element_install.

**Public Members**

**touch_elem_hw_config_t hardware**

Hardware configuration.

**touch_elem_sw_config_t software**

Software configuration.

**struct touch_elem_waterproof_config_t**

Touch element waterproof configuration passed to touch_element_waterproof_install.

**Public Members**

**touch_pad_t guard_channel**

Waterproof Guard-Sensor channel number (index)

**float guard_sensitivity**

Waterproof Guard-Sensor sensitivity.

**struct touch_elem_sleep_config_t**

Touch element sleep configuration passed to touch_element_enable_light_sleep or touch_element_enable_deep_sleep.

**Public Members**

**uint16_t sample_count**

scan times in every measurement, normally equal to the ‘sample_count’ field in ‘touch_elem_hw_config_t’.
**Chapter 2. API Reference**

```c
uint16_t sleep_cycle
    sleep_cycle decide the interval between two measurements, t_sleep = sleep_cycle / (RTC_SLOW_CLK frequency), normally equal to the ‘sleep_cycle’ field in ‘touch_elem_hw_config_t’.
```

**struct touch_elem_message_t**

- Touch element event message type from touch_element_message_receive()

**Public Members**

- `touch_elem_handle_t handle`
  - Touch element handle.

- `touch_elem_type_t element_type`
  - Touch element type.

- `void *arg`
  - User input argument.

- `uint8_t child_msg[8]`
  - Encoded message.

**Macros**

- `TOUCH_ELEM_GLOBAL_DEFAULT_CONFIG()`
- `TOUCH_ELEM_EVENT_NONE`
  - None event.
- `TOUCH_ELEM_EVENT_ON_PRESS`
  - On Press event.
- `TOUCH_ELEM_EVENT_ON_RELEASE`
  - On Release event.
- `TOUCH_ELEM_EVENT_ON_LONGPRESS`
  - On LongPress event.
- `TOUCH_ELEM_EVENT_ON_CALCULATION`
  - On Calculation event.
- `TOUCH_WATERPROOF_GUARD_NOUSE`
  - Waterproof no use guard sensor.

**Type Definitions**

- `typedef void *touch_elem_handle_t`
  - Touch element handle type.

- `typedef uint32_t touch_elem_event_t`
  - Touch element event type.
Enumerations

enum touch_elem_type_t

Touch element handle type.

Values:

enumerator TOUCH_ELEM_TYPE_BUTTON

Touch element button.

enumerator TOUCH_ELEM_TYPE_SLIDER

Touch element slider.

enumerator TOUCH_ELEM_TYPE_MATRIX

Touch element matrix button.

enum touch_elem_dispatch_t

Touch element event dispatch methods (event queue/callback)

Values:

enumerator TOUCH_ELEM_DISP_EVENT

Event queue dispatch.

enumerator TOUCH_ELEM_DISP_CALLBACK

Callback dispatch.

enumerator TOUCH_ELEM_DISP_MAX

API Reference - Touch Button

Header File

- components/touch_element/include/touch_element/touch_button.h
- This header file can be included with:

```c
#include "touch_element/touch_button.h"
```

- This header file is a part of the API provided by the touch_element component. To declare that your component depends on touch_element, add the following to your CMakeLists.txt:

```cmake
Requires touch_element
```

Functions

```c
esp_err_t touch_button_install (const touch_button_global_config_t *global_config)
```

Touch Button initialize.

This function initializes touch button global and acts on all touch button instances.

Parameters: global_config — [in] Button object initialization configuration

Returns:

- ESP_OK: Successfully initialized touch button
- ESP_ERR_INVALID_STATE: Touch element library was not initialized
Chapter 2. API Reference

- ESP_ERR_INVALID_ARG: button_init is NULL
- ESP_ERR_NO_MEM: Insufficient memory

```c
void touch_button_uninstall (void)
```

Release resources allocated using `touch_button_install()`

```c
esp_err_t touch_button_create (const touch_button_config_t *button_config, touch_button_handle_t *button_handle)
```

Create a new touch button instance.

**Note:** The sensitivity has to be explored in experiments, Sensitivity = \( \frac{\text{Raw}(\text{touch}) - \text{Raw}(\text{release})}{\text{Raw}(\text{release})} \times 100\% \)

**Parameters**

- `button_config` - [in] Button configuration
- `button_handle` - [out] Button handle

**Returns**

- ESP_OK: Successfully create touch button
- ESP_ERR_INVALID_STATE: Touch button driver was not initialized
- ESP_ERR_NO_MEM: Insufficient memory
- ESP_ERR_INVALID_ARG: Invalid configuration struct or arguments is NULL

```c
esp_err_t touch_button_delete (touch_button_handle_t button_handle)
```

Release resources allocated using `touch_button_create()`

**Parameters**

- `button_handle` - [in] Button handle

**Returns**

- ESP_OK: Successfully released resources
- ESP_ERR_INVALID_STATE: Touch button driver was not initialized
- ESP_ERR_INVALID_ARG: `button_handle` is null
- ESP_ERR_NOT_FOUND: Input handle is not a button handle

```c
esp_err_t touch_button_subscribe_event (touch_button_handle_t button_handle, uint32_t event_mask, void *arg)
```

Touch button subscribes event.

This function uses event mask to subscribe to touch button events, once one of the subscribed events occurs, the event message could be retrieved by calling `touch_element_message_receive()` or input callback routine.

**Note:** Touch button only support three kind of event masks, they are TOUCH_ELEM_EVENT_ON_PRESS, TOUCH_ELEM_EVENT_ON_RELEASE, TOUCH_ELEM_EVENT_ON_LONGPRESS. You can use those event masks in any combination to achieve the desired effect.

**Parameters**

- `button_handle` - [in] Button handle
- `event_mask` - [in] Button subscription event mask
- `arg` - [in] User input argument

**Returns**

- ESP_OK: Successfully subscribed touch button event
- ESP_ERR_INVALID_STATE: Touch button driver was not initialized
- ESP_ERR_INVALID_ARG: `button_handle` is null or event is not supported

```c
esp_err_t touch_button_set_dispatch_method (touch_button_handle_t button_handle, touch_elem_dispatch_t dispatch_method)
```

Touch button set dispatch method.

This function sets a dispatch method that the driver core will use this method as the event notification method.
### Parameters
- **button_handle** [in] Button handle
- **dispatch_method** [in] Dispatch method (By callback/event)

### Returns
- ESP_OK: Successfully set dispatch method
- ESP_ERR_INVALID_STATE: Touch button driver was not initialized
- ESP_ERR_INVALID_ARG: button_handle is null or dispatch_method is invalid

```c
esp_err_t touch_button_set_callback (touch_button_handle_t button_handle, touch_button_callback_t button_callback)
```

Touch button set callback.

This function sets a callback routine into touch element driver core, when the subscribed events occur, the callback routine will be called.

---

**Note:** Button message will be passed from the callback function and it will be destroyed when the callback function return.

---

**Warning:** Since this input callback routine runs on driver core (esp-timer callback routine), it should not do something that attempts to Block, such as calling vTaskDelay().

### Parameters
- **button_handle** [in] Button handle
- **button_callback** [in] User input callback

### Returns
- ESP_OK: Successfully set callback
- ESP_ERR_INVALID_STATE: Touch button driver was not initialized
- ESP_ERR_INVALID_ARG: button_handle or button_callback is null

```c
esp_err_t touch_button_set_longpress (touch_button_handle_t button_handle, uint32_t threshold_time)
```

Touch button set long press trigger time.

This function sets the threshold time (ms) for a long press event. If a button is pressed and held for a period of time that exceeds the threshold time, a long press event is triggered.

### Parameters
- **button_handle** [in] Button handle
- **threshold_time** [in] Threshold time (ms) of long press event occur

### Returns
- ESP_OK: Successfully set the threshold time of long press event
- ESP_ERR_INVALID_STATE: Touch button driver was not initialized
- ESP_ERR_INVALID_ARG: button_handle is null or time (ms) is not larger than 0

```c
const touch_button_message_t *touch_button_get_message (const touch_elem_message_t *element_message)
```

Touch button get message.

This function decodes the element message from touch_element_message_receive() and return a button message pointer.

### Parameters
- **element_message** [in] element message

### Returns
- Touch button message pointer

---

### Structures
struct touch_button_global_config_t
  Button initialization configuration passed to touch_button_install.

Public Members

float threshold_divider
  Button channel threshold divider.

uint32_t default_lp_time
  Button default LongPress event time (ms)

struct touch_button_config_t
  Button configuration (for new instance) passed to touch_button_create()

Public Members

touch_pad_t channel_num
  Button channel number (index)

float channel_sens
  Button channel sensitivity.

struct touch_button_message_t
  Button message type.

Public Members

touch_button_event_t event
  Button event.

Macros

TOUCH_BUTTON_GLOBAL_DEFAULT_CONFIG()

Type Definitions
typedef touch_elem_handle_t touch_button_handle_t
  Button handle.

typedef void (*touch_button_callback_t)(touch_button_handle_t, touch_button_message_t*, void*)
  Button callback type.

Enumerations
typedef touch_button_event_t
  Button event type.

Values:
enumerator **TOUCH_BUTTON_EVT_ON_PRESS**  
Button Press event.

enumerator **TOUCH_BUTTON_EVT_ON_RELEASE**  
Button Release event.

enumerator **TOUCH_BUTTON_EVT_ON_LONGPRESS**  
Button LongPress event.

enumerator **TOUCH_BUTTON_EVT_MAX**

## API Reference - Touch Slider

### Header File

- components/touch_element/include/touch_element/touch_slider.h
- This header file can be included with:

  ```
  #include "touch_element/touch_slider.h"
  ```

- This header file is a part of the API provided by the touch_element component. To declare that your component depends on touch_element, add the following to your CMakeLists.txt:

  ```
  REQUIRES touch_element
  ```
  
or

  ```
  PRIV_REQUIRES touch_element
  ```

### Functions

**esp_err_t touch_slider_install** (const touch_slider_global_config_t *global_config)

Touch slider initialize.

This function initializes touch slider object and acts on all touch slider instances.

**Parameters**

- `global_config` - [in] Touch slider global initialization configuration

**Returns**

- ESP_OK: Successfully initialized touch slider
- ESP_ERR_INVALID_STATE: Touch element library was not initialized
- ESP_ERR_INVALID_ARG: slider_init is NULL
- ESP_ERR_NO_MEM: Insufficient memory

**void touch_slider_uninstall** (void)

Release resources allocated using touch_slider_install()

**esp_err_t touch_slider_create** (const touch_slider_config_t *slider_config, touch_slider_handle_t *slider_handle)

Create a new touch slider instance.

**Note:** The index of Channel array and sensitivity array must be one-one correspondence

**Parameters**

- `slider_config` - [in] Slider configuration
- `slider_handle` - [out] Slider handle

**Returns**

- ESP_OK: Successfully create touch slider
Chapter 2. API Reference

- ESP_ERR_INVALID_STATE: Touch slider driver was not initialized
- ESP_ERR_INVALID_ARG: Invalid configuration struct or arguments is NULL
- ESP_ERR_NO_MEM: Insufficient memory

**esp_err_t touch_slider_delete(touch_slider_handle_t slider_handle)**

Release resources allocated using touch_slider_create.

**Parameters**
- **slider_handle** - [in] Slider handle

**Returns**
- ESP_OK: Successfully released resources
- ESP_ERR_INVALID_STATE: Touch slider driver was not initialized
- ESP_ERR_INVALID_ARG: slider_handle is null
- ESP_ERR_NOT_FOUND: Input handle is not a slider handle

**esp_err_t touch_slider_subscribe_event(touch_slider_handle_t slider_handle, uint32_t event_mask, void *arg)**

Touch slider subscribes event.

This function uses event mask to subscribe to touch slider events, once one of the subscribed events occurs, the event message could be retrieved by calling touch_element_message_receive() or input callback routine.

**Note:** Touch slider only support three kind of event masks, they are TOUCH_ELEM_EVENT_ON_PRESS, TOUCH_ELEM_EVENT_ON_RELEASE. You can use those event masks in any combination to achieve the desired effect.

**Parameters**
- **slider_handle** - [in] Slider handle
- **event_mask** - [in] Slider subscription event mask
- **arg** - [in] User input argument

**Returns**
- ESP_OK: Successfully subscribed touch slider event
- ESP_ERR_INVALID_STATE: Touch slider driver was not initialized
- ESP_ERR_INVALID_ARG: slider_handle is null or event is not supported

**esp_err_t touch_slider_set_dispatch_method(touch_slider_handle_t slider_handle, touch_elem_dispatch_t dispatch_method)**

Touch slider set dispatch method.

This function sets a dispatch method that the driver core will use this method as the event notification method.

**Parameters**
- **slider_handle** - [in] Slider handle
- **dispatch_method** - [in] Dispatch method (By callback/event)

**Returns**
- ESP_OK: Successfully set dispatch method
- ESP_ERR_INVALID_STATE: Touch slider driver was not initialized
- ESP_ERR_INVALID_ARG: slider_handle is null or dispatch_method is invalid

**esp_err_t touch_slider_set_callback(touch_slider_handle_t slider_handle, touch_slider_callback_t slider_callback)**

Touch slider set callback.

This function sets a callback routine into touch element driver core, when the subscribed events occur, the callback routine will be called.

**Note:** Slider message will be passed from the callback function and it will be destroyed when the callback function return.
Warning: Since this input callback routine runs on driver core (esp-timer callback routine), it should not do something that attempts to Block, such as calling vTaskDelay().

Parameters

- **slider_handle** - [in] Slider handle
- **slider_callback** - [in] User input callback

Returns

- ESP_OK: Successfully set callback
- ESP_ERR_INVALID_STATE: Touch slider driver was not initialized
- ESP_ERR_INVALID_ARG: slider_handle or slider_callback is null

```c
const touch_slider_message_t *touch_slider_get_message (const touch_elem_message_t *element_message)
```

Touch slider get message.

This function decodes the element message from touch_element_message_receive() and return a slider message pointer.

Parameters **element_message** - [in] element message

Returns Touch slider message pointer

Structures

```c
struct touch_slider_global_config_t
```

Slider initialization configuration passed to touch_slider_install.

Public Members

- **float quantify_lower_threshold**
  Slider signal quantification threshold.

- **float threshold_divider**
  Slider channel threshold divider.

- **uint16_t filter_reset_time**
  Slider position filter reset time (Unit is esp_timer callback tick)

- **uint16_t benchmark_update_time**
  Slider benchmark update time (Unit is esp_timer callback tick)

- **uint8_t position_filter_size**
  Moving window filter buffer size.

- **uint8_t position_filter_factor**
  One-order IIR filter factor.

- **uint8_t calculate_channel_count**
  The number of channels which will take part in calculation.

```c
struct touch_slider_config_t
```

Slider configuration (for new instance) passed to touch_slider_create()
Public Members

const touch_pad_t *channel_array
    Slider channel array.

const float *sensitivity_array
    Slider channel sensitivity array.

uint8_t channel_num
    The number of slider channels.

uint8_t position_range
    The right region of touch slider position range, [0, position_range (less than or equal to 255)].

struct touch_slider_message_t
    Slider message type.

Public Members

    touch_slider_event_t event
        Slider event.

    touch_slider_position_t position
        Slider position.

Macros

TOUCH_SLIDER_GLOBAL_DEFAULT_CONFIG()

Type Definitions

typedef uint32_t touch_slider_position_t
    Slider position data type.

typedef touch_elem_handle_t touch_slider_handle_t
    Slider instance handle.

typedef void (*touch_slider_callback_t)(touch_slider_handle_t, touch_slider_message_t*, void*)
    Slider callback type.

Enumerations

gen touch_slider_event_t
    Slider event type.

    Values:

        enumerator TOUCH_SLIDER_EVT_ON_PRESS
            Slider on Press event.
enumerator **TOUCH_SLIDER_EVT_ON_RELEASE**
Slider on Release event.

enumerator **TOUCH_SLIDER_EVT_ON_CALCULATION**
Slider on Calculation event.

enumerator **TOUCH_SLIDER_EVT_MAX**

### API Reference - Touch Matrix

#### Header File

- components/touch_element/include/touch_element/touch_matrix.h
- This header file can be included with:
  ```c
  #include "touch_element/touch_matrix.h"
  ```
- This header file is a part of the API provided by the `touch_element` component. To declare that your component depends on `touch_element`, add the following to your CMakeLists.txt:
  ```cmake
  REQUIRES touch_element
  ```
  or
  ```cmake
  PRIV_REQUIRES touch_element
  ```

#### Functions

- **esp_err_t touch_matrix_install** (const `touch_matrix_global_config_t *global_config`)
  Touch matrix button initialize.
  This function initializes touch matrix button object and acts on all touch matrix button instances.

  **Parameters**
  - `global_config` - [in] Touch matrix global initialization configuration

  **Returns**
  - ESP_OK: Successfully initialized touch matrix button
  - ESP_ERR_INVALID_STATE: Touch element library was not initialized
  - ESP_ERR_INVALID_ARG: matrix_init is NULL
  - ESP_ERR_NO_MEM: Insufficient memory

- **void touch_matrix_uninstall** (void)
  Release resources allocated using `touch_matrix_install()`

- **esp_err_t touch_matrix_create** (const `touch_matrix_config_t *matrix_config, touch_matrix_handle_t *matrix_handle`)
  Create a new touch matrix button instance.

  **Note:** Channel array and sensitivity array must be one-one correspondence in those array

  **Note:** Touch matrix button does not support Multi-Touch now

  **Parameters**
  - `matrix_config` - [in] Matrix button configuration
  - `matrix_handle` - [out] Matrix button handle

  **Returns**
  - ESP_OK: Successfully create touch matrix button
Chapter 2. API Reference

- ESP_ERR_INVALID_STATE: Touch matrix driver was not initialized
- ESP_ERR_INVALID_ARG: Invalid configuration struct or arguments is NULL
- ESP_ERR_NO_MEM: Insufficient memory

```c
esp_err_t touch_matrix_delete(touch_matrix_handle_t matrix_handle)
```

Release resources allocated using touch_matrix_create()

Parameters
- `matrix_handle` - [in] Matrix handle

Returns
- ESP_OK: Successfully released resources
- ESP_ERR_INVALID_STATE: Touch matrix driver was not initialized
- ESP_ERR_INVALID_ARG: matrix_handle is null
- ESP_ERR_NOT_FOUND: Input handle is not a matrix handle

```c
esp_err_t touch_matrix_subscribe_event(touch_matrix_handle_t matrix_handle,uint32_t event_mask,
void *arg)
```

Touch matrix button subscribes event.

This function uses event mask to subscribe to touch matrix events, once one of the subscribed events occurs, the event message could be retrieved by calling touch_element_message_receive() or input callback routine.

Note: Touch matrix button only support three kind of event masks, they are TOUCH_ELEM_EVENT_ON_PRESS, TOUCH_ELEM_EVENT_ON_RELEASE, TOUCH_ELEM_EVENT_ON_LONGPRESS. You can use those event masks in any combination to achieve the desired effect.

Parameters
- `matrix_handle` - [in] Matrix handle
- `event_mask` - [in] Matrix subscription event mask
- `arg` - [in] User input argument

Returns
- ESP_OK: Successfully subscribed touch matrix event
- ESP_ERR_INVALID_STATE: Touch matrix driver was not initialized
- ESP_ERR_INVALID_ARG: matrix_handle is null or event is not supported

```c
esp_err_t touch_matrix_set_dispatch_method(touch_matrix_handle_t matrix_handle,
touch_elem_dispatch_t dispatch_method)
```

Touch matrix button set dispatch method.

This function sets a dispatch method that the driver core will use this method as the event notification method.

Parameters
- `matrix_handle` - [in] Matrix button handle
- `dispatch_method` - [in] Dispatch method (By callback/event)

Returns
- ESP_OK: Successfully set dispatch method
- ESP_ERR_INVALID_STATE: Touch matrix driver was not initialized
- ESP_ERR_INVALID_ARG: matrix_handle is null or dispatch_method is invalid

```c
esp_err_t touch_matrix_set_callback(touch_matrix_handle_t matrix_handle, touch_matrix_callback_f matrix_callback)
```

Touch matrix button set callback.

This function sets a callback routine into touch element driver core, when the subscribed events occur, the callback routine will be called.

Note: Matrix message will be passed from the callback function and it will be destroyed when the callback function return.
**Warning:** Since this input callback routine runs on driver core (esp-timer callback routine), it should not do something that attempts to block, such as calling vTaskDelay().

Parameters
- `matrix_handle` - [in] Matrix button handle
- `matrix_callback` - [in] User input callback

Returns
- ESP_OK: Successfully set callback
- ESP_ERR_INVALID_STATE: Touch matrix driver was not initialized
- ESP_ERR_INVALID_ARG: `matrix_handle` or `matrix_callback` is null

```c
esp_err_t touch_matrix_set_longpress(touch_matrix_handle_t matrix_handle, uint32_t threshold_time)
```

Touch matrix button set long press trigger time.

This function sets the threshold time (ms) for a long press event. If a matrix button is pressed and held for a period of time that exceeds the threshold time, a long press event is triggered.

Parameters
- `matrix_handle` - [in] Matrix button handle
- `threshold_time` - [in] Threshold time (ms) of long press event occur

Returns
- ESP_OK: Successfully set the time of long press event
- ESP_ERR_INVALID_STATE: Touch matrix driver was not initialized
- ESP_ERR_INVALID_ARG: `matrix_handle` is null or time (ms) is 0

```c
const touch_matrix_message_t *touch_matrix_get_message(const touch_elem_message_t *element_message)
```

Touch matrix get message.

This function decodes the element message from `touch_element_message_receive()` and return a matrix message pointer.

Parameters `element_message` - [in] element message

Returns Touch matrix message pointer

**Structures**

```c
struct touch_matrix_global_config_t
```

Matrix button initialization configuration passed to `touch_matrix_install`.

**Public Members**

```c
float threshold_divider
```

Matrix button channel threshold divider.

```c
uint32_t default_lp_time
```

Matrix button default LongPress event time (ms)

```c
struct touch_matrix_config_t
```

Matrix button configuration (for new instance) passed to `touch_matrix_create()`

**Public Members**
const touch_pad_t *x_channel_array
Matrix button x-axis channels array.

const touch_pad_t *y_channel_array
Matrix button y-axis channels array.

const float *x_sensitivity_array
Matrix button x-axis channels sensitivity array.

const float *y_sensitivity_array
Matrix button y-axis channels sensitivity array.

uint8_t x_channel_num
The number of channels in x-axis.

uint8_t y_channel_num
The number of channels in y-axis.

struct touch_matrix_position_t
Matrix button position data type.

Public Members

uint8_t x_axis
Matrix button x axis position.

uint8_t y_axis
Matrix button y axis position.

uint8_t index
Matrix button position index.

struct touch_matrix_message_t
Matrix message type.

Public Members

touch_matrix_event_t event
Matrix event.

touch_matrix_position_t position
Matrix position.

Macros
TOUCH_MATRIX_GLOBAL_DEFAULT_CONFIG()
Type Definitions

typedef touch_elem_handle_t touch_matrix_handle_t
        Matrix button instance handle.

typedef void (*touch_matrix_callback_t)(touch_matrix_handle_t, touch_matrix_message_t*, void*)
        Matrix button callback type.

Enumerations

enum touch_matrix_event_t
        Matrix button event type.
        Values:

        enumerator TOUCH_MATRIX_EVT_ON_PRESS
                Matrix button Press event.

        enumerator TOUCH_MATRIX_EVT_ON_RELEASE
                Matrix button Press event.

        enumerator TOUCH_MATRIX_EVT_ON_LONGPRESS
                Matrix button LongPress event.

        enumerator TOUCH_MATRIX_EVT_MAX

2.5.26 Two-Wire Automotive Interface (TWAI)

Overview

The Two-Wire Automotive Interface (TWAI) is a real-time serial communication protocol suited for automotive
and industrial applications. It is compatible with ISO11898-1 Classical frames, thus can support Standard Frame
Format (11-bit ID) and Extended Frame Format (29-bit ID). The ESP32-S2 contains 1 TWAI controller(s) that can
be configured to communicate on a TWAI bus via an external transceiver.

Warning: The TWAI controller is not compatible with ISO11898-1 FD Format frames, and will interpret such
frames as errors.

This programming guide is split into the following sections:

Sections

- Two-Wire Automotive Interface (TWAI)
  - Overview
  - TWAI Protocol Summary
  - Signals Lines and Transceiver
  - API Naming Conventions
  - Driver Configuration
  - Driver Operation
  - Examples
  - API Reference
TWAI Protocol Summary

The TWAI is a multi-master, multi-cast, asynchronous, serial communication protocol. TWAI also supports error detection and signalling, and inbuilt message prioritization.

**Multi-master:** Any node on the bus can initiate the transfer of a message.

**Multi-cast:** When a node transmits a message, all nodes on the bus will receive the message (i.e., broadcast) thus ensuring data consistency across all nodes. However, some nodes can selectively choose which messages to accept via the use of acceptance filtering (multi-cast).

**Asynchronous:** The bus does not contain a clock signal. All nodes on the bus operate at the same bit rate and synchronize using the edges of the bits transmitted on the bus.

**Error Detection and Signaling:** Every node constantly monitors the bus. When any node detects an error, it signals the detection by transmitting an error frame. Other nodes will receive the error frame and transmit their own error frames in response. This results in an error detection being propagated to all nodes on the bus.

**Message Priorities:** Messages contain an ID field. If two or more nodes attempt to transmit simultaneously, the node transmitting the message with the lower ID value will win arbitration of the bus. All other nodes will become receivers ensuring that there is at most one transmitter at any time.

**TWAI Messages**  
TWAI Messages are split into Data Frames and Remote Frames. Data Frames are used to deliver a data payload to other nodes, whereas a Remote Frame is used to request a Data Frame from other nodes (other nodes can optionally respond with a Data Frame). Data and Remote Frames have two frame formats known as **Extended Frame** and **Standard Frame** which contain a 29-bit ID and an 11-bit ID respectively. A TWAI message consists of the following fields:

- 29-bit or 11-bit ID: Determines the priority of the message (lower value has higher priority).
- Data Length Code (DLC) between 0 to 8: Indicates the size (in bytes) of the data payload for a Data Frame, or the amount of data to request for a Remote Frame.
- Up to 8 bytes of data for a Data Frame (should match DLC).

**Error States and Counters**  
The TWAI protocol implements a feature known as “fault confinement” where a persistently erroneous node will eventually eliminate itself from the bus. This is implemented by requiring every 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 incremented and decremented according to a set of rules (where the counters increase on an error, and decrease on a successful message transmission/reception). The values of the counters are used to determine a node’s error state, namely Error Active, Error Passive, and Bus-Off.

**Error Active:** A node is Error Active when both TEC and REC are less than 128 and indicates that the node is operating normally. Error Active nodes are allowed to participate in bus communications, and will actively signal the detection of any errors by automatically transmitting an Active Error Flag over the bus.

**Error Passive:** A 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 bus communications, but will instead transmit a Passive Error Flag upon detection of an error.

**Bus-Off:** A node becomes Bus-Off when the TEC becomes greater than or equal to 256. A Bus-Off node is unable influence the bus in any manner (essentially disconnected from the bus) thus eliminating itself from the bus. A node will remain in the Bus-Off state until it undergoes bus-off recovery.

**Signals Lines and Transceiver**

The TWAI controller does not contain an integrated transceiver. Therefore, to connect the TWAI controller to a TWAI bus, an external transceiver is required. The type of external transceiver used should depend on the application’s physical layer specification (e.g., using SN65HVD23x transceivers for ISO 11898-2 compatibility).

The TWAI 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 the ESP32-S2’s GPIO pads.
**TX and RX:** The TX and RX signal lines are required to interface with an external transceiver. Both signal lines represent/interpret a dominant bit as a low logic level (0 V), and a recessive bit as a high logic level (3.3 V).

**BUS-OFF:** The BUS-OFF signal line is optional and is set to a low logic level (0 V) whenever the TWAI controller reaches a bus-off state. The BUS-OFF signal line is set to a high logic level (3.3 V) otherwise.

**CLKOUT:** The CLKOUT signal line is optional and outputs a prescaled version of the controller’s source clock.

**Note:** An external transceiver must internally loopback the TX to RX 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 TWAI controller to interpret differences in logic levels between the two signal lines as a loss in arbitration or a bit error.

**API Naming Conventions**

**Note:** The TWAI driver provides two sets of API. One is handle-free and is widely used in IDF versions earlier than v5.2, but it can only support one TWAI hardware controller. The other set is with handles, and the function name is usually suffixed with “v2”, which can support any number of TWAI controllers. These two sets of API can be used at the same time, but it is recommended to use the “v2” version in your new projects.

**Driver Configuration**

This section covers how to configure the TWAI driver.

**Operating Modes** The TWAI driver supports the following modes of operations:

**Normal Mode:** The normal operating mode allows the TWAI controller to take part in bus activities such as transmitting and receiving messages/error frames. Acknowledgement from another node is required when transmitting a message.

**No Ack Mode:** The No Acknowledgement mode is similar to normal mode, however acknowledgements are not required for a message transmission to be considered successful. This mode is useful when self testing the TWAI controller (loopback of transmissions).
Listen Only Mode: This mode prevents the TWAI controller from influencing the bus. Therefore, transmission of messages/acknowledgement/error frames will be disabled. However, the TWAI controller is still able to receive messages but will not acknowledge the message. This mode is suited for bus monitor applications.

Alerts The TWAI driver contains an alert feature that is used to notify the application layer of certain TWAI controller or TWAI bus events. Alerts are selectively enabled when the TWAI driver is installed, but can be reconfigured during runtime by calling `twai_reconfigure_alerts()`. The application can then wait for any enabled alerts to occur by calling `twai_read_alerts()`. The TWAI driver supports the following alerts:

<table>
<thead>
<tr>
<th>Alert Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWAI_ALERT_TX_IDLE</td>
<td>No more messages queued for transmission</td>
</tr>
<tr>
<td>TWAI_ALERT_TX_SUCCESS</td>
<td>The previous transmission was successful</td>
</tr>
<tr>
<td>TWAI_ALERT_RX_DATA</td>
<td>A frame has been received and added to the RX queue</td>
</tr>
<tr>
<td>TWAI_ALERT_BELOW_ERR_WARN</td>
<td>Both error counters have dropped below error warning limit</td>
</tr>
<tr>
<td>TWAI_ALERT_ERR_ACTIVE</td>
<td>TWAI controller has become error active</td>
</tr>
<tr>
<td>TWAI_ALERT_RECOVERY_IN_PROGRESS</td>
<td>TWAI controller is undergoing bus recovery</td>
</tr>
<tr>
<td>TWAI_ALERT_BUS_RECOVERED</td>
<td>TWAI controller has successfully completed bus recovery</td>
</tr>
<tr>
<td>TWAI_ALERT_ARB_LOST</td>
<td>The previous transmission lost arbitration</td>
</tr>
<tr>
<td>TWAI_ALERT ABOVE_ERR_WARN</td>
<td>One of the error counters have exceeded the error warning limit</td>
</tr>
<tr>
<td>TWAI_ALERT BUS ERROR</td>
<td>A (Bit, Stuff, CRC, Form, ACK) error has occurred on the bus</td>
</tr>
<tr>
<td>TWAI_ALERT_TX_FAILED</td>
<td>The previous transmission has failed</td>
</tr>
<tr>
<td>TWAI_ALERT RX_QUEUE_FULL</td>
<td>The RX queue is full causing a received frame to be lost</td>
</tr>
<tr>
<td>TWAI_ALERT_ERR_PASS</td>
<td>TWAI controller has become error passive</td>
</tr>
<tr>
<td>TWAI_ALERT_BUS_OFF</td>
<td>Bus-off condition occurred. TWAI controller can no longer influence bus</td>
</tr>
</tbody>
</table>

Note: The TWAI controller’s error warning limit is used to preemptively warn the application of bus errors before the error passive state is reached. By default, the TWAI driver sets the error warning limit to 96. The TWAI_ALERT_ABV_ERR_WARN is raised when the TEC or REC becomes larger than or equal to the error warning limit. The TWAI_ALERT_BELOW_ERR_WARN is raised when both TEC and REC return back to values below 96.

Note: When enabling alerts, the TWAI_ALERT_AND_LOG flag can be used to cause the TWAI driver to log any raised alerts to UART. However, alert logging is disabled and TWAI_ALERT_AND_LOG if the CONFIG_TWAI_ISR_IN_IRAM option is enabled (see Placing ISR into IRAM).

Note: The TWAI_ALERT_ALL and TWAI_ALERT_NONE macros can also be used to enable/disable all alerts during configuration/reconfiguration.

Bit Timing The operating bit rate of the TWAI driver is configured using the `twai_timing_config_t` structure. The period of each bit is made up of multiple time quanta, and the period of a time quantum is determined by a pre-scaled version of the TWAI controller’s source clock. A single bit contains the following segments in the following order:

1. The Synchronization Segment consists of a single time quantum
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 quantum by dividing the TWAI controller’s source clock. On the ESP32-S2, the `brp` can be any even number from 2 to 32768. Alternatively, you can decide
the resolution of each quantum, by setting `twai_timing_config_t::quanta_resolution_hz` to a non-zero value. In this way, the driver can calculate the underlying `brp` value for you. It is useful when you set different clock sources but want the bitrate to keep the same.

Supported clock source for a TWAI controller is listed in the `twai_clock_source_t` and can be specified in `twai_timing_config_t::clk_src`.

![Bit timing configuration for 500kbit/s given BRP = 8, clock source frequency is 80MHz](image)

Fig. 30: Bit timing configuration for 500kbit/s given BRP = 8, clock source frequency is 80MHz

The sample point of a bit is located on the intersection of Timing Segment 1 and 2. Enabling **Triple Sampling** causes 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 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 bit rates. The following macro initializers are provided by the TWAI driver.

- `TWAI_TIMING_CONFIG_1MBITS`
- `TWAI_TIMING_CONFIG_800KBITS`
- `TWAI_TIMING_CONFIG_500KBITS`
- `TWAI_TIMING_CONFIG_250KBITS`
- `TWAI_TIMING_CONFIG_125KBITS`
- `TWAI_TIMING_CONFIG_100KBITS`
- `TWAI_TIMING_CONFIG_50KBITS`
- `TWAI_TIMING_CONFIG_25KBITS`
- `TWAI_TIMING_CONFIG_20KBITS`
- `TWAI_TIMING_CONFIG_16KBITS`
- `TWAI_TIMING_CONFIG_12_5KBITS`
- `TWAI_TIMING_CONFIG_10KBITS`
- `TWAI_TIMING_CONFIG_5KBITS`
- `TWAI_TIMING_CONFIG_1KBITS`

**Acceptance Filter** The TWAI controller contains a hardware acceptance filter which can be used to filter messages of a particular ID. A node that filters out a message does not receive the message, but will still acknowledge it. Acceptance filters can make a node more efficient by filtering out messages sent over the bus that are irrelevant to the node. The acceptance filter is configured using two 32-bit values within `twai_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 TWAI controller. The **acceptance mask** is a bit sequence specifying which bits of the acceptance code can be ignored. This allows for a messages of different IDs to be accepted by a single acceptance code.
Chapter 2. API Reference

The acceptance filter can be used under **Single or Dual Filter Mode**. Single Filter Mode uses 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 are interpreted under Single Filter Mode (Note: The yellow and blue fields represent standard and extended frame formats respectively).

![Fig. 31: Bit layout of single filter mode (Right side MSBit)](image)

**Dual Filter Mode** uses 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 are interpreted under **Dual Filter Mode** (Note: The yellow and blue fields represent standard and extended frame formats respectively).

![Fig. 32: Bit layout of dual filter mode (Right side MSBit)](image)

**Disabling TX Queue** The TX queue can be disabled during configuration by setting the `tx_queue_len` member of `twai_general_config_t` to 0. This allows applications that do not require message transmission to save a small amount of memory when using the TWAI driver.

**Placing ISR into IRAM** The TWAI 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 TWAI driver’s functionality during lengthy cache disabling operations (such as SPI Flash writes, OTA updates etc). Whilst the cache is disabled, the ISR continues 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 TWAI driver’s ISR, users must do the following:

- Enable the `CONFIG_TWAI_ISR_IN_IRAM` option using `idf.py menuconfig`.
- When calling `twai_driver_install()`, the `intr_flags` member of `twai_general_config_t` should set the `ESP_INTR_FLAG_IRAM` set.

**Note:** When the `CONFIG_TWAI_ISR_IN_IRAM` option is enabled, the TWAI driver will no longer log any alerts (i.e., the TWAI_ALERT_AND_LOG flag will not have any effect).

**Driver Operation**

The TWAI 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. 33: State transition diagram of the TWAI driver (see table below)**

<table>
<thead>
<tr>
<th>Label</th>
<th>Transition</th>
<th>Action/Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Uninstalled &gt; Stopped</td>
<td><code>twai_driver_install()</code></td>
</tr>
<tr>
<td>B</td>
<td>Stopped &gt; Uninstalled</td>
<td><code>twai_driver_uninstall()</code></td>
</tr>
<tr>
<td>C</td>
<td>Stopped &gt; Running</td>
<td><code>twai_start()</code></td>
</tr>
<tr>
<td>D</td>
<td>Running &gt; Stopped</td>
<td><code>twai_stop()</code></td>
</tr>
<tr>
<td>E</td>
<td>Running &gt; Bus-Off</td>
<td>Transmit Error Counter &gt;= 256</td>
</tr>
<tr>
<td>F</td>
<td>Bus-Off &gt; Uninstalled</td>
<td><code>twai_driver_uninstall()</code></td>
</tr>
<tr>
<td>G</td>
<td>Bus-Off &gt; Recovering</td>
<td><code>twai_initiate_recovery()</code></td>
</tr>
<tr>
<td>H</td>
<td>Recovering &gt; Stopped</td>
<td>128 occurrences of 11 consecutive recessive bits.</td>
</tr>
</tbody>
</table>

**Driver States**

**Uninstalled:** In the uninstalled state, no memory is allocated for the driver and the TWAI controller is powered OFF.

**Stopped:** In this state, the TWAI controller is powered ON and the TWAI driver has been installed. However the TWAI controller is unable to take part in any bus activities such as transmitting, receiving, or acknowledging messages.

**Running:** In the running state, the TWAI controller is able to take part in bus activities. Therefore messages can be transmitted/received/acknowledged. Furthermore, the TWAI controller is able to transmit error frames upon detection of errors on the bus.

**Bus-Off:** The bus-off state is automatically entered when the TWAI controller’s Transmit Error Counter becomes greater than or equal to 256. The bus-off state indicates the occurrence of severe errors on the bus or in the TWAI controller. Whilst in the bus-off state, the TWAI controller is unable to take part in any bus activities. To exit the bus-off state, the TWAI controller must undergo the bus recovery process.

**Recovering:** The recovering state is entered when the TWAI controller undergoes bus recovery. The TWAI controller/TWAI driver remains in the recovering state until the 128 occurrences of 11 consecutive recessive bits is observed on the bus.

**Message Fields and Flags** The TWAI driver distinguishes different types of messages by using the various bit field members of the `twai_message_t` structure. These bit field members determine whether a message is in standard or extended format, a remote frame, 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 `twai_message_t` and the following message flags:
## Message Flag

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWAI_MSG_FLAG_EXTD</td>
<td>Message is in Extended Frame Format (29bit ID)</td>
</tr>
<tr>
<td>TWAI_MSG_FLAG_RTR</td>
<td>Message is a Remote Frame (Remote Transmission Request)</td>
</tr>
<tr>
<td>TWAI_MSG_FLAG_SS</td>
<td>Transmit message using Single Shot Transmission (Message will not be re-transmitted upon error or loss of arbitration). Unused for received message.</td>
</tr>
<tr>
<td>TWAI_MSG_FLAG_SELF</td>
<td>Transmit message using Self Reception Request (Transmitted message will also be received by the same node). Unused for received message.</td>
</tr>
<tr>
<td>TWAI_MSG_FLAG_DLC_NON_COMP</td>
<td>Message’s Data length code is larger than 8. This will break compliance with TWAI</td>
</tr>
<tr>
<td>TWAI_MSG_FLAG_NONE</td>
<td>Clears all bit fields. Equivalent to a Standard Frame Format (11bit ID) Data Frame.</td>
</tr>
</tbody>
</table>

### Examples

#### Configuration & Installation

The following code snippet demonstrates how to configure, install, and start the TWAI driver via the use of the various configuration structures, macro initializers, the `twai_driver_install()` function, and the `twai_start()` function.

```c
#include "driver/gpio.h"
#include "driver/twai.h"

void app_main()
{
    //Initialize configuration structures using macro initializers
    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_21, GPIO_NUM_22, TWAI_MODE_NORMAL);
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();

    //Install TWAI driver
    if (twai_driver_install(&g_config, &t_config, &f_config) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    //Start TWAI driver
    if (twai_start() == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    ...
}
```

The usage of macro initializers is not mandatory and each of the configuration structures can be manually.

#### Install Multiple TWAI Instances

The following code snippet demonstrates how to install multiple TWAI instances via the use of the `twai_driver_install_v2()` function.

```c
#include "driver/gpio.h"
#include "driver/twai.h"

void app_main()
{
    // Install multiple TWAI instances
    int num_instances = 2;
    int i;
    for (i = 0; i < num_instances; i++) {
        // Initialize and install configurations
        // ...
    }
    // Start the TWAI driver
    if (twai_start() == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }
}
```

(continues on next page)
Chapter 2. API Reference

(continued from previous page)

```c
{
    twai_handle_t twai_bus_0;
    twai_handle_t twai_bus_1;
    // Initialize configuration structures using macro initializers
    twai_general_config_t g_config = TWAI_GENERAL_CONFIG_DEFAULT(GPIO_NUM_0, GPIO_NUM_1, TWAI_MODE_NORMAL);
    twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
    twai_filter_config_t f_config = TWAI_FILTER_CONFIG_ACCEPT_ALL();

    // Install driver for TWAI bus 0
    g_config.controller_id = 0;
    if (twai_driver_install_v2(&g_config, &t_config, &f_config, &twai_bus_0) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    // Start TWAI driver
    if (twai_start_v2(twai_bus_0) == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    // Install driver for TWAI bus 1
    g_config.controller_id = 1;
    g_config.tx_io = GPIO_NUM_2;
    g_config.rx_io = GPIO_NUM_3;
    if (twai_driver_install_v2(&g_config, &t_config, &f_config, &twai_bus_1) == ESP_OK) {
        printf("Driver installed\n");
    } else {
        printf("Failed to install driver\n");
        return;
    }

    // Start TWAI driver
    if (twai_start_v2(twai_bus_1) == ESP_OK) {
        printf("Driver started\n");
    } else {
        printf("Failed to start driver\n");
        return;
    }

    // Other Driver operations must use version 2 API as well
    ...
}
```

**Message Transmission** The following code snippet demonstrates how to transmit a message via the usage of the `twai_message_t` type and `twai_transmit()` function.

```c
#include "driver/twai.h"
...
// Configure message to transmit
twai_message_t message;
message.identifier = 0xAAAA;
```

(continues on next page)
message.extd = 1;
message.data_length_code = 4;
for (int i = 0; i < 4; i++) {
    message.data[i] = 0;
}

//Queue message for transmission
if (twai_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 twai_message_t type and twai_receive() function.

#include "driver/twai.h"
...

//Wait for message to be received
twai_message_t message;
if (twai_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 TWAI driver alerts via the use of the twai_reconfigure_alerts() and twai_read_alerts() functions.

#include "driver/twai.h"
...

//Reconfigure alerts to detect Error Passive and Bus-Off error states
uint32_t alerts_to_enable = TWAI_ALERT_ERR_PASS | TWAI_ALERT_BUS_OFF;
if (twai_reconfigure_alerts(alerts_to_enable, NULL) == ESP_OK) {
    printf("Alerts reconfigured\n");
} else {
    printf("Failed to reconfigure alerts\n");
}

//Block indefinitely until an alert occurs
uint32_t alerts_triggered;
twai_read_alerts(&alerts_triggered, portMAX_DELAY);

Stop and Uninstall The following code demonstrates how to stop and uninstall the TWAI driver via the use of the `twai_stop()` and `twai_driver_uninstall()` functions.

```c
#include "driver/twai.h"
...

//Stop the TWAI driver
if (twai_stop() == ESP_OK) {
    printf("Driver stopped\n");
} else {
    printf("Failed to stop driver\n");
    return;
}

//Uninstall the TWAI driver
if (twai_driver_uninstall() == ESP_OK) {
    printf("Driver uninstalled\n");
} else {
    printf("Failed to uninstall driver\n");
    return;
}
```

Multiple ID Filter Configuration The acceptance mask in `twai_filter_config_t` can be configured such that two or more IDs are 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 the 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 TWAI Network example demonstrates communication between two ESP32-S2s using the TWAI driver API. One TWAI node acts as a network master that initiates and ceases the transfer of data from another node acting as a network slave. The example can be found via `peripherals/twai/twai_network`.

Alert and Recovery Example: This example demonstrates how to use the TWAI driver's alert and bus-off recovery API. The example purposely introduces errors on the bus to put the TWAI 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/twai/twai_alert_and_recovery`.

Self Test Example: This example uses the No Acknowledge Mode and Self Reception Request to cause the TWAI controller to send and simultaneously receive a series of messages. This example can be used to verify if the connections between the TWAI controller and the external transceiver are working correctly. The example can be found via `peripherals/twai/twai_self_test`.

API Reference

Header File
Chapter 2. API Reference

- components/hal/include/hal/twai_types.h
- This header file can be included with:

```c
#include "hal/twai_types.h"
```

Structures

```c
struct twai_message_t
```
Structure to store a TWAI message.

**Note:** The flags member is deprecated

Public Members

```c
uint32_t extd
```
Extended Frame Format (29bit ID)

```c
uint32_t rtr
```
Message is a Remote Frame

```c
uint32_t ss
```
Transmit as a Single Shot Transmission. Unused for received.

```c
uint32_t self
```
Transmit as a Self Reception Request. Unused for received.

```c
uint32_t dlc_non_comp
```
Message’s Data length code is larger than 8. This will break compliance with ISO 11898-1

```c
uint32_t reserved
```
Reserved bits

```c
uint32_t flags
```
Deprecated: Alternate way to set bits using message flags

```c
uint32_t identifier
```
11 or 29 bit identifier

```c
uint8_t data_length_code
```
Data length code

```c
uint8_t data[TWAI_FRAME_MAX_DLC]
```
Data bytes (not relevant in RTR frame)

```c
struct twai_timing_config_t
```
Structure for bit timing configuration of the TWAI driver.

**Note:** Macro initializers are available for this structure
Public Members

`twai_clock_source_t clk_src`
Clock source, set to 0 or TWAI_CLK_SRC_DEFAULT if you want a default clock source

`uint32_t quanta_resolution_hz`
The resolution of one timing quanta, in Hz. Note: the value of `brp` will be reflected by this field if it’s non-zero, otherwise, `brp` needs to be set manually

`uint32_t brp`
Baudrate prescale (i.e., clock divider). Any even number from 2 to 128 for ESP32, 2 to 32768 for non-ESP32 chip. Note: For ESP32 ECO 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 TWAI controller samples a bit

`struct twai_filter_config_t`
Structure for acceptance filter configuration of the TWAI 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**

`TWAI_EXTD_ID_MASK`
TWAI Constants.
Bit mask for 29 bit Extended Frame Format ID
TWAI_STD_ID_MASK
Bit mask for 11 bit Standard Frame Format ID

TWAI_FRAME_MAX_DLC
Max data bytes allowed in TWAI

TWAI_FRAME_EXTD_ID_LEN_BYTES
EFF ID requires 4 bytes (29bit)

TWAI_FRAME_STD_ID_LEN_BYTES
SFF ID requires 2 bytes (11bit)

TWAI_ERR_PASS_THRESH
Error counter threshold for error passive

Type Definitions
typedef soc_periph_twai_clk_src_t twai_clock_source_t
RMT group clock source.

Note: User should select the clock source based on the power and resolution requirement

Enumerations
enum twai_mode_t
TWAI Controller operating modes.
Values:

enumerator TWAI_MODE_NORMAL
Normal operating mode where TWAI controller can send/receive/acknowledge messages

enumerator TWAI_MODE_NO_ACK
Transmission does not require acknowledgment. Use this mode for self testing

enumerator TWAI_MODE_LISTEN_ONLY
The TWAI controller will not influence the bus (No transmissions or acknowledgments) but can receive messages

Header File
• components/driver/twai/include/driver/twai.h
• This header file can be included with:

```
#include "driver/twai.h"
```

• This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

```
REQUIRES driver
```
or
Functions

\texttt{esp_err_t \twai\_driver\_install} (const \texttt{twai\_general\_config\_t} *\texttt{g\_config}, const \texttt{twai\_timing\_config\_t} *\texttt{t\_config}, const \texttt{twai\_filter\_config\_t} *\texttt{f\_config})

Install TWAI driver.

This function installs the TWAI driver using three configuration structures. The required memory is allocated and the TWAI driver is placed in the stopped state after running this function.

\textbf{Note: } Macro initializers are available for the configuration structures (see documentation)

\textbf{Note: } To reinstall the TWAI driver, call \texttt{twai\_driver\_uninstall()} first

Parameters

\begin{itemize}
  \item \texttt{g\_config} [in] General configuration structure
  \item \texttt{t\_config} [in] Timing configuration structure
  \item \texttt{f\_config} [in] Filter configuration structure
\end{itemize}

Returns

\begin{itemize}
  \item ESP_OK: Successfully installed TWAI driver
  \item ESP_ERR_INVALID_ARG: Arguments are invalid, e.g. invalid clock source, invalid quanta resolution
  \item ESP_ERR_NO_MEM: Insufficient memory
  \item ESP_ERR_INVALID_STATE: Driver is already installed
\end{itemize}

\texttt{esp_err_t \twai\_driver\_install\_v2} (const \texttt{twai\_general\_config\_t} *\texttt{g\_config}, const \texttt{twai\_timing\_config\_t} *\texttt{t\_config}, const \texttt{twai\_filter\_config\_t} *\texttt{f\_config}, \texttt{twai\_handle\_t} \texttt{*ret\_twai})

Install TWAI driver and return a handle.

\textbf{Note: } This is an advanced version of \texttt{twai\_driver\_install} that can return a driver handle, so that it allows you to install multiple TWAI drivers. Don’ t forget to set the proper controller\_id in the \texttt{twai\_general\_config\_t}. Please refer to the documentation of \texttt{twai\_driver\_install\_v2} for more details.

Parameters

\begin{itemize}
  \item \texttt{g\_config} [in] General configuration structure
  \item \texttt{t\_config} [in] Timing configuration structure
  \item \texttt{f\_config} [in] Filter configuration structure
  \item \texttt{ret\_twai} [out] Pointer to a new created TWAI handle
\end{itemize}

Returns

\begin{itemize}
  \item ESP_OK: Successfully installed TWAI driver
  \item ESP_ERR_INVALID_ARG: Arguments are invalid, e.g. invalid clock source, invalid quanta resolution, invalid controller ID
  \item ESP_ERR_NO_MEM: Insufficient memory
  \item ESP_ERR_INVALID_STATE: Driver is already installed
\end{itemize}

\texttt{esp_err_t \twai\_driver\_uninstall} (void)

Uninstall the TWAI driver.

This function uninstalls the TWAI 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.

Returns
- ESP_OK: Successfully uninstalled TWAI driver
- ESP_ERR_INVALID_STATE: Driver is not in stopped/bus-off state, or is not installed

```c
esp_err_t twai_driver_uninstall_v2(twai_handle_t handle)
```
Uninstall the TWAI driver with a given handle.

Note: This is an advanced version of `twai_driver_uninstall` that can uninstall a TWAI driver with a given handle. Please refer to the documentation of `twai_driver_uninstall` for more details.

Parameters
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`

Returns
- ESP_OK: Successfully uninstalled TWAI driver
- ESP_ERR_INVALID_STATE: Driver is not in stopped/bus-off state, or is not installed

```c
esp_err_t twai_start(void)
```
Start the TWAI driver.

This function starts the TWAI driver, putting the TWAI driver into the running state. This allows the TWAI driver to participate in TWAI bus activities such as transmitting/receiving messages. The TX and RX queue are reset in this function, clearing any messages that are unread or pending transmission. This function can only be called when the TWAI driver is in the stopped state.

Returns
- ESP_OK: TWAI driver is now running
- ESP_ERR_INVALID_STATE: Driver is not in stopped state, or is not installed

```c
esp_err_t twai_start_v2(twai_handle_t handle)
```
Start the TWAI driver with a given handle.

Note: This is an advanced version of `twai_start` that can start a TWAI driver with a given handle. Please refer to the documentation of `twai_start` for more details.

Parameters
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`

Returns
- ESP_OK: TWAI driver is now running
- ESP_ERR_INVALID_STATE: Driver is not in stopped state, or is not installed

```c
esp_err_t twai_stop(void)
```
Stop the TWAI driver.

This function stops the TWAI driver, preventing any further message from being transmitted or received until `twai_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 TWAI driver is in the running state.

Warning: A message currently being transmitted/received on the TWAI bus will be ceased immediately. This may lead to other TWAI nodes interpreting the unfinished message as an error.

Returns
Chapter 2. API Reference

- ESP_OK: TWAI driver is now Stopped
- ESP_ERR_INVALID_STATE: Driver is not in running state, or is not installed

`esp_err_t twai_stop_v2 (twai_handle_t handle)`

Stop the TWAI driver with a given handle.

**Note:** This is an advanced version of `twai_stop` that can stop a TWAI driver with a given handle. Please refer to the documentation of `twai_stop` for more details.

**Parameters**
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`

**Returns**
- ESP_OK: TWAI driver is now Stopped
- ESP_ERR_INVALID_STATE: Driver is not in running state, or is not installed

`esp_err_t twai_transmit (const twai_message_t *message, TickType_t ticks_to_wait)`

Transmit a TWAI message.

This function queues a TWAI 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 times out. 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 TWAI 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.

**Parameters**
- `message` - [in] Message to transmit
- `ticks_to_wait` - [in] Number of FreeRTOS ticks to block on the TX queue

**Returns**
- 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: TWAI driver is not in running state, or is not installed
- ESP_ERR_NOT_SUPPORTED: Listen Only Mode does not support transmissions

`esp_err_t twai_transmit_v2 (twai_handle_t handle, const twai_message_t *message, TickType_t ticks_to_wait)`

Transmit a TWAI message via a given handle.

**Note:** This is an advanced version of `twai_transmit` that can transmit a TWAI message with a given handle. Please refer to the documentation of `twai_transmit` for more details.

**Parameters**
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`
- `message` - [in] Message to transmit
- `ticks_to_wait` - [in] Number of FreeRTOS ticks to block on the TX queue

**Returns**
Chapter 2. API Reference

- 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: TWAI driver is not in running state, or is not installed
- ESP_ERR_NOT_SUPPORTED: Listen Only Mode does not support transmissions

```c
esp_err_t twai_receive(twai_message_t *message, TickType_t ticks_to_wait)
```

Receive a TWAI 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.

**Parameters**
- message - [out] Received message
- ticks_to_wait – [in] Number of FreeRTOS ticks to block on RX queue

**Returns**
- 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: TWAI driver is not installed

```c
esp_err_t twai_receive_v2(twai_handle_t handle, twai_message_t *message, TickType_t ticks_to_wait)
```

Receive a TWAI message via a given handle.

**Note:** This is an advanced version of `twai_receive` that can receive a TWAI message with a given handle. Please refer to the documentation of `twai_receive` for more details.

**Parameters**
- handle - [in] TWAI driver handle returned by `twai_driver_install_v2`
- message - [out] Received message
- ticks_to_wait – [in] Number of FreeRTOS ticks to block on RX queue

**Returns**
- 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: TWAI driver is not installed

```c
esp_err_t twai_read_alerts(uint32_t *alerts, TickType_t ticks_to_wait)
```

Read TWAI driver alerts.

This function will read the alerts raised by the TWAI driver. If no alert has been issued 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.

**Parameters**
- alerts - [out] Bit field of raised alerts (see documentation for alert flags)
- ticks_to_wait – [in] Number of FreeRTOS ticks to block for alert

**Returns**
Chapter 2. API Reference

- ESP_OK: Alerts read
- ESP_ERR_TIMEOUT: Timed out waiting for alerts
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

`esp_err_t twai_read_alerts_v2 (twai_handle_t handle, uint32_t *alerts, TickType_t ticks_to_wait)`

Read TWAI driver alerts with a given handle.

**Note:** This is an advanced version of `twai_read_alerts` that can read TWAI driver alerts with a given handle. Please refer to the documentation of `twai_read_alerts` for more details.

**Parameters**
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`
- `alerts` - [out] Bit field of raised alerts (see documentation for alert flags)
- `ticks_to_wait` - [in] Number of FreeRTOS ticks to block for alert

**Returns**
- ESP_OK: Alerts read
- ESP_ERR_TIMEOUT: Timed out waiting for alerts
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

`esp_err_t twai_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.

**Parameters**
- `alerts_enabled` - [in] Bit field of alerts to enable (see documentation for alert flags)
- `current_alerts` - [out] Bit field of currently raised alerts. Set to NULL if unused

**Returns**
- ESP_OK: Alerts reconfigured
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

`esp_err_t twai_reconfigure_alerts_v2 (twai_handle_t handle, uint32_t alerts_enabled, uint32_t *current_alerts)`

Reconfigure which alerts are enabled, with a given handle.

**Note:** This is an advanced version of `twai_reconfigure_alerts` that can reconfigure which alerts are enabled with a given handle. Please refer to the documentation of `twai_reconfigure_alerts` for more details.

**Parameters**
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`
- `alerts_enabled` - [in] Bit field of alerts to enable (see documentation for alert flags)
- `current_alerts` - [out] Bit field of currently raised alerts. Set to NULL if unused

**Returns**
- ESP_OK: Alerts reconfigured
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

`esp_err_t twai_initiate_recovery (void)`

Start the bus recovery process.

This function initiates the bus recovery process when the TWAI driver is in the bus-off state. Once initiated, the TWAI driver will enter the recovering state and wait for 128 occurrences of the bus-free signal on the
TWAI 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.

### Returns
- ESP_OK: Bus recovery started
- ESP_ERR_INVALID_STATE: TWAI driver is not in the bus-off state, or is not installed

```c
esp_err_t twai_initiate_recovery_v2 (twai_handle_t handle)
```
Start the bus recovery process with a given handle.

**Note:** This is an advanced version of `twai_initiate_recovery` that can start the bus recovery process with a given handle. Please refer to the documentation of `twai_initiate_recovery` for more details.

### Parameters
- **handle**  
  [in] TWAI driver handle returned by `twai_driver_install_v2`

### Returns
- ESP_OK: Bus recovery started
- ESP_ERR_INVALID_STATE: TWAI driver is not in the bus-off state, or is not installed

```c
esp_err_t twai_get_status_info (twai_status_info_t *status_info)
```
Get current status information of the TWAI driver.

**Parameters**
- **status_info**  
  [out] Status information

### Returns
- ESP_OK: Status information retrieved
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

```c
esp_err_t twai_get_status_info_v2 (twai_handle_t handle, twai_status_info_t *status_info)
```
Get current status information of a given TWAI driver handle.

**Note:** This is an advanced version of `twai_get_status_info` that can get current status information of a given TWAI driver handle. Please refer to the documentation of `twai_get_status_info` for more details.

### Parameters
- **handle**  
  [in] TWAI driver handle returned by `twai_driver_install_v2`
- **status_info**  
  [out] Status information

### Returns
- ESP_OK: Status information retrieved
- ESP_ERR_INVALID_ARG: Arguments are invalid
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

```c
esp_err_t twai_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 `twai_stop()` or `twai_initiate_recovery()` is called.

### Returns
Chapter 2. API Reference

- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed or TX queue is disabled

`esp_err_t twai_clear_transmit_queue_v2 (twai_handle_t handle)`

Clear the transmit queue of a given TWAI driver handle.

Note: This is an advanced version of `twai_clear_transmit_queue` that can clear the transmit queue of a given TWAI driver handle. Please refer to the documentation of `twai_clear_transmit_queue` for more details.

Parameters
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`

Returns
- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed or TX queue is disabled

`esp_err_t twai_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 `twai_start()` is called.

Returns
- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

`esp_err_t twai_clear_receive_queue_v2 (twai_handle_t handle)`

Clear the receive queue of a given TWAI driver handle.

Note: This is an advanced version of `twai_clear_receive_queue` that can clear the receive queue of a given TWAI driver handle. Please refer to the documentation of `twai_clear_receive_queue` for more details.

Parameters
- `handle` - [in] TWAI driver handle returned by `twai_driver_install_v2`

Returns
- ESP_OK: Transmit queue cleared
- ESP_ERR_INVALID_STATE: TWAI driver is not installed

Structures

`struct twai_general_config_t`

Structure for general configuration of the TWAI driver.

Note: Macro initializers are available for this structure

Public Members
**controller_id**

TWAI controller ID, index from 0. If you want to install TWAI driver with a non-zero controller_id, please use `twai_driver_install_v2`.

**mode**

Mode of TWAI controller

**tx_io**

Transmit GPIO number

**rx_io**

Receive GPIO number

**clkout_io**

CLKOUT GPIO number (optional, set to -1 if unused)

**bus_off_io**

Bus off indicator GPIO number (optional, set to -1 if unused)

**tx_queue_len**

Number of messages TX queue can hold (set to 0 to disable TX Queue)

**rx_queue_len**

Number of messages RX queue can hold

**alerts_enabled**

Bit field of alerts to enable (see documentation)

**clkout_divider**

CLKOUT divider. Can be 1 or any even number from 2 to 14 (optional, set to 0 if unused)

**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.

**twai_status_info_t**

Structure to store status information of TWAI driver.

### Public Members

**state**

Current state of TWAI controller (Stopped/Running/Bus-Off/Recovery)

**msgs_to_tx**

Number of messages queued for transmission or awaiting transmission completion

**msgs_to_rx**

Number of messages in RX queue waiting to be read
Chapter 2. API Reference

```c
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 (or errata workaround if enabled)

uint32_t rx_overrun_count
    Number of messages that were lost due to a RX FIFO overrun

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**

```
TWAI_IO_UNUSED
    Marks GPIO as unused in TWAI configuration
```

**Type Definitions**

```c
typedef struct twai_obj_t *twai_handle_t
    TWAI controller handle.
```

**Enumerations**

```
enum twai_state_t
    TWAI driver states.

Values:

enumerator TWAI_STATE_STOPPED
    Stopped state. The TWAI controller will not participate in any TWAI bus activities

enumerator TWAI_STATE_RUNNING
    Running state. The TWAI controller can transmit and receive messages

enumerator TWAI_STATE_BUS_OFF
    Bus-off state. The TWAI controller cannot participate in bus activities until it has recovered

enumerator TWAI_STATE_RECOVERING
    Recovering state. The TWAI controller is undergoing bus recovery
```

Espressif Systems 834

Submit Document Feedback
2.5.27 Universal Asynchronous Receiver/Transmitter (UART)

Introduction

A Universal Asynchronous Receiver/Transmitter (UART) is a hardware feature that handles communication (i.e., timing requirements and data framing) using widely-adopted asynchronous serial communication interfaces, such as RS232, RS422, and RS485. A UART provides a widely adopted and cheap method to realize full-duplex or half-duplex data exchange among different devices.

The ESP32-S2 chip has 2 UART controllers (also referred to as port), each featuring an identical set of registers to simplify programming and for more 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 regular UART controllers are compatible with UART-enabled devices from various manufacturers and can also support Infrared Data Association (IrDA) protocols.

Functional Overview

The overview describes how to establish communication between an ESP32-S2 and other UART devices using the functions and data types of the UART driver. A typical programming workflow is broken down into the sections provided below:

1. **Set Communication Parameters** - Setting baud rate, data bits, stop bits, etc.
2. **Set Communication Pins** - Assigning pins for connection to a device
3. **Install Drivers** - Allocating ESP32-S2’s resources for the UART driver
4. **Run UART Communication** - Sending/receiving data
5. **Use 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 UART controllers using `uart_port_t`. This identification is needed for all the following function calls.

**Set 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.

```c
const uart_port_t uart_num = UART_NUM_1;
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));
```

For more information on how to configure the hardware flow control options, please refer to `peripherals/uart/uart_echo`.

**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 5: Functions for Configuring specific parameters individually

<table>
<thead>
<tr>
<th>Parameter to Configure</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baud rate</td>
<td>uart_set_baudrate()</td>
</tr>
<tr>
<td>Number of transmitted bits</td>
<td>uart_set_word_length() selected out of uart_word_length_t</td>
</tr>
<tr>
<td>Parity control</td>
<td>uart_set_parity() selected out of uart_parity_t</td>
</tr>
<tr>
<td>Number of stop bits</td>
<td>uart_set_stop_bits() selected out of uart_stop_bits_t</td>
</tr>
<tr>
<td>Hardware flow control mode</td>
<td>uart_set_hw_flow_ctrl() selected out of uart_hw_flowcontrol_t</td>
</tr>
<tr>
<td>Communication mode</td>
<td>uart_set_mode() selected out of uart_mode_t</td>
</tr>
</tbody>
</table>

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()`.

**Set 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 `UART_PIN_NO_CHANGE` should be specified for pins that will not be used.

```
// Set UART pins (TX: IO4, RX: IO5, RTS: IO18, CTS: IO19)
ESP_ERROR_CHECK(uart_set_pin(UART_NUM_1, 4, 5, 18, 19));
```

**Install Drivers**  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 allocates the required internal 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_1, 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.

**Run 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 only writes and reads data from a specific buffer using `uart_write_bytes()` and `uart_read_bytes()` respectively, and the FSM does the rest.
Transmit Data  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 copies 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.

```c
// 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.

```c
// 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 does not block until space is available. Instead, it writes 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.

```c
// Wait for packet to be sent
const uart_port_t uart_num = UART_NUM_1;
ESP_ERROR_CHECK(uart_wait_tx_done(uart_num, 100)); // wait timeout is 100 RTOS_ticks (TickType_t)
```

Receive Data  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.

```c
// Read data from UART.
const uart_port_t uart_num = UART_NUM_1;
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 handles 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.

```c
// Setup UART in rs485 half duplex mode
ESP_ERROR_CHECK(uart_set_mode(uart_num, UART_MODE_RS485_HALF_DUPLEX));
```

Use Interrupts  There are many interrupts that can be generated depending on specific UART states or detected errors. The full list of available interrupts is provided in ESP32-S2 Technical Reference Manual > UART Controller.
(UART) > UART Interrupts and UHCI Interrupts [PDF]. You can enable or disable specific interrupts by calling `uart_enable_intr_mask()` or `uart_disable_intr_mask()` respectively.

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).

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 Install Drivers. 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. This functionality is demonstrated in the example peripherals/uart/uart_events. It can be used, e.g., to detect a command string with a specific number of identical characters (the ‘pattern’) at the end. The following functions are available:
  - Configure and enable this interrupt using `uart_enable_pattern_det_baud_intr()`;
  - Disable the interrupt using `uart_disable_pattern_det_intr()`

**Macros** The API also defines several macros. For example, `UART_HW_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 uses `[UART_REGISTER_NAME].[UART_FIELD_BIT]` to refer to UART register fields/bits. For more information on a specific option bit, see ESP32-S2 Technical Reference Manual > UART Controller (UART) > Register Summary [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-S2’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-S2 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-S2’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**

```
VCC  +---------------+  |
      |               |
      R           B   |
RXD  +-------x-------+  |
      |               |
      D  ADM483 | RS485 bus side |
TXD  |       |       |       |
      |       |       |       |
      DE  A     /RE   |
      +-----+  +-----+  +-----+  +-----+
      |    |    |    |    |    |    |
      GND GND GND GND GND GND
```

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 UART peripheral when the bit UART_RS485_CONF_REG.UART_RS485TX_RX_EN is enabled.

**Circuit B: Manual Switching Transmitter/Receiver Without Collision Detection**

```
VCC  +---------------+  |
      |               |
      R           B   |
RXD  +-------x-------+  |
      |               |
      D  ADM483 | RS485 bus side |
TXD  |       |       |       |
      |       |       |       |
      DE  A     /RE   |
      +-----+  +-----+  +-----+  +-----+
      |    |    |    |    |    |    |
      GND GND GND GND GND GND
```

(continues on next page)
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

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/.

<table>
<thead>
<tr>
<th>Code Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>peripherals/uart/uart_echo</td>
<td>Configuring UART settings, installing the UART driver, and reading/writing over the UART1 interface.</td>
</tr>
<tr>
<td>peripherals/uart/uart_events</td>
<td>Reporting various communication events, using pattern detection interrupts.</td>
</tr>
<tr>
<td>peripherals/uart_async_rxtxtasks</td>
<td>Transmitting and receiving data in two separate FreeRTOS tasks over the same UART.</td>
</tr>
<tr>
<td>peripherals/uart/uart_select</td>
<td>Using synchronous I/O multiplexing for UART file descriptors.</td>
</tr>
<tr>
<td>peripherals/uart/uart_echo_rs485</td>
<td>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-S2 pins.</td>
</tr>
<tr>
<td>peripherals/uart/nmea0183_parser</td>
<td>Obtaining GPS information by parsing NMEA0183 statements received from GPS via the UART peripheral.</td>
</tr>
</tbody>
</table>

API Reference

Header File

Espressif Systems
Chapter 2. API Reference

- components/driver/uart/include/driver/uart.h

  This header file can be included with:

  ```
  #include "driver/uart.h"
  ```

  This header file is a part of the API provided by the driver component. To declare that your component depends on driver, add the following to your CMakeLists.txt:

  ```
  REQUIRES driver
  or
  PRIV_REQUIRES driver
  ```

**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_HW_FIFO_LEN(uart_num). Tx_buffer_size should be either zero or greater than UART_HW_FIFO_LEN(uart_num).

**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)

**Returns**

- ESP_OK Success
- ESP_FAIL Parameter error

```
esp_err_t uart_driver_delete (uart_port_t uart_num)
```

Uninstall UART driver.

**Parameters**

- **uart_num** – UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**

- ESP_OK Success
- ESP_FAIL Parameter error

```
bool uart_is_driver_installed (uart_port_t uart_num)
```

Checks whether the driver is installed or not.

**Parameters**

- **uart_num** – UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**

- true driver is installed
- false driver is not installed

```
esp_err_t uart_set_word_length (uart_port_t uart_num, uart_word_length_t data_bit)
```

Set UART data bits.

**Parameters**
• uart_num - UART port number, the max port number is (UART_NUM_MAX-1).
• data_bit - UART data bits

Returns
• ESP_OK Success
• ESP_FAIL Parameter error

`esp_err_t uart_get_word_length (uart_port_t uart_num, uart_word_length_t *data_bit)`

Get the UART data bit configuration.

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.

Returns
• ESP_FAIL Parameter error
• ESP_OK Success, result will be put in (*data_bit)

`esp_err_t uart_set_stop_bits (uart_port_t uart_num, uart_stop_bits_t stop_bits)`

Set UART stop bits.

Parameters
• uart_num - UART port number, the max port number is (UART_NUM_MAX-1).
• stop_bits - UART stop bits

Returns
• ESP_OK Success
• ESP_FAIL Fail

`esp_err_t uart_get_stop_bits (uart_port_t uart_num, uart_stop_bits_t *stop_bits)`

Get the UART stop bit configuration.

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.

Returns
• ESP_FAIL Parameter error
• ESP_OK Success, result will be put in (*stop_bit)

`esp_err_t uart_set_parity (uart_port_t uart_num, uart_parity_t parity_mode)`

Set UART parity mode.

Parameters
• uart_num - UART port number, the max port number is (UART_NUM_MAX-1).
• parity_mode - the enum of uart parity configuration

Returns
• ESP_FAIL Parameter error
• ESP_OK Success

`esp_err_t uart_get_parity (uart_port_t uart_num, uart_parity_t *parity_mode)`

Get the UART parity mode configuration.

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.

Returns
• ESP_FAIL Parameter error
• ESP_OK Success

`esp_err_t uart_get_sclk_freq (uart_sclk_t sclk, uint32_t *out_freq_hz)`

Get the frequency of a clock source for the HP UART port.

Parameters
• sclk - Clock source
• out_freq_hz - [out] Output of frequency, in Hz

Returns
• ESP_ERR_INVALID_ARG: if the clock source is not supported
• otherwise ESP_OK

\texttt{esp_err_t uart_set_baudrate(uart_port_t uart_num, uint32_t baudrate)}
Set UART baud rate.

\textbf{Parameters}
• \texttt{uart_num} - UART port number, the max port number is (UART_NUM_MAX -1).
• \texttt{baudrate} - UART baud rate.

\textbf{Returns}
• ESP_FAIL Parameter error
• ESP_OK Success

\texttt{esp_err_t uart_get_baudrate(uart_port_t uart_num, uint32_t *baudrate)}
Get the UART baud rate configuration.

\textbf{Parameters}
• \texttt{uart_num} - UART port number, the max port number is (UART_NUM_MAX -1).
• \texttt{baudrate} - Pointer to accept value of UART baud rate

\textbf{Returns}
• ESP_FAIL Parameter error
• ESP_OK Success, result will be put in (*baudrate)

\texttt{esp_err_t uart_set_line_inverse(uart_port_t uart_num, uint32_t inverse_mask)}
Set UART line inverse mode.

\textbf{Parameters}
• \texttt{uart_num} - UART port number, the max port number is (UART_NUM_MAX -1).
• \texttt{inverse_mask} - Choose the wires that need to be inverted. Using the ORred mask of \texttt{uart_signal_inv_t}

\textbf{Returns}
• ESP_OK Success
• ESP_FAIL Parameter error

\texttt{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.

\textbf{Parameters}
• \texttt{uart_num} - UART port number, the max port number is (UART_NUM_MAX -1).
• \texttt{flow_ctrl} - Hardware flow control mode
• \texttt{rx_thresh} - Threshold of Hardware RX flow control (0 ~ UART_HW_FIFO_LEN(uart_num)). Only when UART_HW_FLOWCTRL_RTS is set, will the rx_thresh value be set.

\textbf{Returns}
• ESP_OK Success
• ESP_FAIL Parameter error

\texttt{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.

\textbf{Parameters}
• \texttt{uart_num} - UART port number, the max port number is (UART_NUM_MAX -1)
• \texttt{enable} - switch on or off
• \texttt{rx_thresh_xon} - low water mark
• \texttt{rx_thresh_xoff} - high water mark

\textbf{Returns}
• ESP_OK Success
• ESP_FAIL Parameter error
Chapter 2. API Reference

`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.

**Parameters**
- `uart_num` – UART port number, the max port number is (UART_NUM_MAX -1).
- `flow_ctrl` – Option for different flow control mode.

**Returns**
- ESP_FAIL Parameter error
- ESP_OK Success, result will be put in (*flow_ctrl)

`esp_err_t uart_clear_intr_status(uart_port_t uart_num, uint32_t clr_mask)`

Clear UART interrupt status.

**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.

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

`esp_err_t uart_enable_intr_mask(uart_port_t uart_num, uint32_t enable_mask)`

Set UART interrupt enable.

**Parameters**
- `uart_num` – UART port number, the max port number is (UART_NUM_MAX -1).
- `enable_mask` – Bit mask of the enable bits.

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

`esp_err_t uart_disable_intr_mask(uart_port_t uart_num, uint32_t disable_mask)`

Clear UART interrupt enable bits.

**Parameters**
- `uart_num` – UART port number, the max port number is (UART_NUM_MAX -1).
- `disable_mask` – Bit mask of the disable bits.

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

`esp_err_t uart_enable_rx_intr(uart_port_t uart_num)`

Enable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

**Parameters**
- `uart_num` – UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

`esp_err_t uart_disable_rx_intr(uart_port_t uart_num)`

Disable UART RX interrupt (RX_FULL & RX_TIMEOUT INTERRUPT)

**Parameters**
- `uart_num` – UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

`esp_err_t uart_disable_tx_intr(uart_port_t uart_num)`

Disable UART TX interrupt (TX_FULL & TX_TIMEOUT INTERRUPT)

**Parameters**
- `uart_num` – UART port number

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error
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)

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_HW_FIFO_LEN(uart_num)

Returns
• ESP_OK Success
• ESP_FAIL Parameter error

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)
Assign signals of a UART peripheral to GPIO pins.

Note: If the GPIO number configured for a UART signal matches one of the IOMUX signals for that GPIO, the signal will be connected directly via the IOMUX. Otherwise the GPIO and signal will be connected via the GPIO Matrix. For example, if on an ESP32 the call uart_set_pin(0, 1, 3, -1, -1) is performed, as GPIO1 is UART0’s default TX pin and GPIO3 is UART0’s default RX pin, both will be connected to respectively U0TXD and U0RXD through the IOMUX, totally bypassing the GPIO matrix. The check is performed on a per-pin basis. Thus, it is possible to have RX pin binded to a GPIO through the GPIO matrix, whereas TX is binded to its GPIO through the IOMUX.

Note: Internal signal can be output to multiple GPIO pads. Only one GPIO pad can connect with input signal.

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.

Returns
• ESP_OK Success
• ESP_FAIL Parameter error

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.

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)

Returns
• ESP_OK Success
• ESP_FAIL Parameter error

esp_err_t uart_set_dtr(uart_port_t uart_num, int level)
Manually set the UART DTR pin level.

Parameters
• uart_num – UART port number, the max port number is (UART_NUM_MAX -1).
• level – 1: DTR output low; 0: DTR output high

Returns
• ESP_OK Success
• ESP_FAIL Parameter error

```c
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.

**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)

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

```c
esp_err_t uart_param_config(uart_port_t uart_num, const uart_config_t *uart_config)
```
Set UART configuration parameters.

**Parameters**
- `uart_num` - UART port number, the max port number is (UART_NUM_MAX -1).
- `uart_config` - UART parameter settings

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

```c
esp_err_t uart_intr_config(uart_port_t uart_num, const uart_intr_config_t *intr_conf)
```
Configure UART interrupts.

**Parameters**
- `uart_num` - UART port number, the max port number is (UART_NUM_MAX -1).
- `intr_conf` - UART interrupt settings

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

```c
esp_err_t uart_wait_tx_done(uart_port_t uart_num, TickType_t ticks_to_wait)
```
Wait until UART TX FIFO is empty.

**Parameters**
- `uart_num` - UART port number, the max port number is (UART_NUM_MAX -1).
- `ticks_to_wait` - Timeout, count in RTOS ticks

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error
- ESP_ERR_TIMEOUT Timeout

```c
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.

**Parameters**
- `uart_num` - UART port number, the max port number is (UART_NUM_MAX -1).
- `buffer` - data buffer address
- `len` - data length to send

**Returns**
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO
### uart_write_bytes

**Declaration:**
```
int uart_write_bytes(uart_port_t uart_num, const void *src, size_t size)
```

**Description:**
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.

**Parameters**
- `uart_num` - UART port number, the max port number is (UART_NUM_MAX -1).
- `src` - data buffer address
- `size` - data length to send

**Returns**
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

### uart_write_bytes_with_break

**Declaration:**
```
int uart_write_bytes_with_break(uart_port_t uart_num, const void *src, size_t size, int brk_len)
```

**Description:**
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.

**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)

**Returns**
- (-1) Parameter error
- OTHERS (>=0) The number of bytes pushed to the TX FIFO

### uart_read_bytes

**Declaration:**
```
int uart_read_bytes(uart_port_t uart_num, void *buf, uint32_t length, TickType_t ticks_to_wait)
```

**Description:**
UART read bytes from UART buffer.

**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

**Returns**
- (-1) Error
- OTHERS (>=0) The number of bytes read from UART buffer

### uart_flush

**Declaration:**
```
esp_err_t uart_flush(uart_port_t uart_num)
```

**Description:**
Alias of uart_flush_input. UART ring buffer flush. This will discard all data in the UART RX buffer.

**Parameters**
- `uart_num` - UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

---

**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.
**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.

---

**Parameters**
- **uart_num** – UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

**esp_err_t uart_get_buffered_data_len (uart_port_t uart_num, size_t *size)**
UART get RX ring buffer cached data length.

**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

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

**esp_err_t uart_get_tx_buffer_free_size (uart_port_t uart_num, size_t *size)**
UART get TX ring buffer free space size.

**Parameters**
- **uart_num** – UART port number, the max port number is (UART_NUM_MAX -1).
- **size** – Pointer of size_t to accept the free space size

**Returns**
- ESP_OK Success
- ESP_ERR_INVALID_ARG Parameter error

**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.

**Parameters**
- **uart_num** – UART port number, the max port number is (UART_NUM_MAX -1).

**Returns**
- ESP_OK Success
- ESP_FAIL Parameter error

**esp_err_t uart_enable_pattern_det_baud_intr (uart_port_t uart_num, char patternChr, 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.

**Parameters**
- **uart_num** – UART port number.
- **patternChr** – 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.

**Returns**
- ESP_OK Success
• ESP_FAIL Parameter error

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.

Parametersuart_num – UART port number, the max port number is (UART_NUM_MAX-1).
Returns
• (-1) No pattern found for current index or parameter error
• others the pattern position in rx buffer.

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.

Parametersuart_num – UART port number, the max port number is (UART_NUM_MAX-1).
Returns
• (-1) No pattern found for current index or parameter error
• others the pattern position in rx buffer.

esp_err_tuart_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.

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.

Returns
• ESP_ERR_NO_MEM No enough memory
• ESP_ERR_INVALID_STATE Driver not installed
• ESP_FAIL Parameter error
• ESP_OK Success

esp_err_tuart_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.
Parameters
• uart_num – Uart number to configure, the max port number is (UART_NUM_MAX -1).
• mode – UART UART mode to set

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

`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

Parameters
• uart_num – UART port number, the max port number is (UART_NUM_MAX -1)
• threshold – Threshold value above which RX fifo full interrupt is generated

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error
• ESP_ERR_INVALID_STATE Driver is not installed

`esp_err_t uart_set_tx_empty_threshold (uart_port_t uart_num, int threshold)`
Set uart threshold values for TX fifo empty.

Parameters
• uart_num – UART port number, the max port number is (UART_NUM_MAX -1)
• threshold – Threshold value below which TX fifo empty interrupt is generated

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error
• ESP_ERR_INVALID_STATE Driver is not installed

`esp_err_t uart_set_rx_timeout (uart_port_t uart_num, const uint8_t tout_thresh)`
UART set threshold timeout for TOUT feature.

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.

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error
• ESP_ERR_INVALID_STATE Driver is not installed

`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()).

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.

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

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 ensure that UART has correct Baud rate all the time, it is necessary to select a source clock which has a fixed frequency and remains active during sleep. For the supported clock sources of the chips, please refer to uart_sclk_t or soc_periph_uart_clk_src_legacy_t

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.

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.

Returns
• ESP_OK on success
• ESP_ERR_INVALID_ARG if uart_num is incorrect or wakeup_threshold is outside of [3, 0x3ff] range.

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.

Parameters
• uart_num – UART number, the max port number is (UART_NUM_MAX - 1).
• out_wakeup_threshold – [out] output, set to the current value of wakeup threshold for the given UART.

Returns
• ESP_OK on success
• ESP_ERR_INVALID_ARG if out_wakeup_threshold is NULL

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).

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.

* •

**Returns**
- 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_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
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_PIN_NO_CHANGE**

**UART_FIFO_LEN**

Length of the HP UART HW FIFO.

UART_HW_FIFO_LEN(uart_num)

Length of the UART HW FIFO.

**UART_BITRATE_MAX**

Maximum configurable bitrate.
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:

enumerator UART_DATA

UART data event

enumerator UART_BREAK

UART break event

enumerator UART_BUFFER_FULL

UART RX buffer full event

enumerator UART_FIFO_OVF

UART FIFO overflow event

enumerator UART_FRAME_ERR

UART RX frame error event

enumerator UART_PARITY_ERR

UART RX parity event

enumerator UART_DATA_BREAK

UART TX data and break event

enumerator UART_PATTERN_DET

UART pattern detected

enumerator UART_WAKEUP

UART wakeup event

enumerator UART_EVENT_MAX

UART event max index

Header File

- components/hal/include/hal/uart_types.h
- This header file can be included with:

```c
#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

Type Definitions

typedef soc_periph_uart_clk_src_legacy_t uart_sclk_t
 UART source clock.

Enumerations

enum uart_port_t
 UART port number, can be UART_NUM_0 ~ (UART_NUM_MAX -1).

 Values:

 enumerator UART_NUM_0
 UART port 0
Chapter 2. API Reference

enumerator UART_NUM_1
UART port 1

enumerator UART_NUM_MAX
UART port max

enum uart_mode_t
UART mode selection.

Values:

enumerator UART_MODE_UART
mode: regular UART mode

enumerator UART_MODE_RS485_HALF_DUPLEX
mode: half duplex RS485 UART mode control by RTS pin

enumerator UART_MODE_IRDA
mode: IRDA UART mode

enumerator UART_MODE_RS485_COLLISION_DETECT
mode: RS485 collision detection UART mode (used for test purposes)

enumerator UART_MODE_RS485_APP_CTRL
mode: application control RS485 UART mode (used for test purposes)

enum uart_word_length_t
UART word length constants.

Values:

enumerator UART_DATA_5_BITS
word length: 5 bits

enumerator UART_DATA_6_BITS
word length: 6 bits

enumerator UART_DATA_7_BITS
word length: 7 bits

enumerator UART_DATA_8_BITS
word length: 8 bits

enumerator UART_DATA_BITS_MAX

enum uart_stop_bits_t
UART stop bits number.

Values:
Chapter 2. API Reference

enumerator UART_STOP_BITS_1
    stop bit: 1bit

enumerator UART_STOP_BITS_1_5
    stop bit: 1.5bits

enumerator UART_STOP_BITS_2
    stop bit: 2bits

enumerator UART_STOP_BITS_MAX

enum uart_parity_t
    UART parity constants.
    Values:
        enumerator UART_PARITY_DISABLE
            Disable UART parity
        enumerator UART_PARITY_EVEN
            Enable UART even parity
        enumerator UART_PARITY_ODD
            Enable UART odd parity

enum uart_hw_flowcontrol_t
    UART hardware flow control modes.
    Values:
        enumerator UART_HW_FLOWCTRL_DISABLE
            disable hardware flow control
        enumerator UART_HW_FLOWCTRL_RTS
            enable RX hardware flow control (rts)
        enumerator UART_HW_FLOWCTRL_CTS
            enable TX hardware flow control (cts)
        enumerator UART_HW_FLOWCTRL_CTS_RTS
            enable hardware flow control
        enumerator UART_HW_FLOWCTRL_MAX

enum uart_signal_inv_t
    UART signal bit map.
    Values:
        enumerator UART_SIGNAL_INV_DISABLE
            Disable UART signal inverse
enumerator UART_SIGNAL_IRDA_TX_INV
    inverse the UART irda_tx signal

enumerator UART_SIGNAL_IRDA_RX_INV
    inverse the UART irda_rx signal

enumerator UART_SIGNAL_RXD_INV
    inverse the UART rxd signal

enumerator UART_SIGNAL_CTS_INV
    inverse the UART cts signal

enumerator UART_SIGNAL_DSR_INV
    inverse the UART dsr signal

enumerator UART_SIGNAL_TXD_INV
    inverse the UART txd signal

enumerator UART_SIGNAL_RTS_INV
    inverse the UART rts signal

enumerator UART_SIGNAL_DTR_INV
    inverse the UART dtr signal

**GPIOLookup 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 returns 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). It is similar to the above macro but specifies the pin function which is also part of the IO_MUX assignment.

**Header File**

- components/soc/esp32s2/include/soc/uart_channel.h
- This header file can be included with:

```
#include "soc/uart_channel.h"
```
Macros

- UART_GPIO43_DIRECT_CHANNEL
- UART_NUM_0_TXD_DIRECT_GPIO_NUM
- UART_GPIO44_DIRECT_CHANNEL
- UART_NUM_0_RXD_DIRECT_GPIO_NUM
- UART_GPIO16_DIRECT_CHANNEL
- UART_NUM_0_CTS_DIRECT_GPIO_NUM
- UART_GPIO15_DIRECT_CHANNEL
- UART_NUM_0_RTS_DIRECT_GPIO_NUM
- UART_TXD_GPIO43_DIRECT_CHANNEL
- UART_RXD_GPIO44_DIRECT_CHANNEL
- UART_CTS_GPIO16_DIRECT_CHANNEL
- UART_RTS_GPIO15_DIRECT_CHANNEL
- UART_GPIO17_DIRECT_CHANNEL
- UART_NUM_1_TXD_DIRECT_GPIO_NUM
- UART_GPIO18_DIRECT_CHANNEL
- UART_NUM_1_RXD_DIRECT_GPIO_NUM
- UART_GPIO20_DIRECT_CHANNEL
- UART_NUM_1_CTS_DIRECT_GPIO_NUM
- UART_GPIO19_DIRECT_CHANNEL
- UART_NUM_1_RTS_DIRECT_GPIO_NUM
- UART_TXD_GPIO17_DIRECT_CHANNEL
- UART_RXD_GPIO18_DIRECT_CHANNEL
- UART_CTS_GPIO20_DIRECT_CHANNEL
UART_RTS_GPIO19_DIRECT_CHANNEL

2.5.28 USB Device Driver

Overview

The driver allows you to use ESP32-S2 chips to develop USB devices on a top of TinyUSB stack. TinyUSB is integrated with ESP-IDF to provide USB features of the framework. Using this driver the chip works as simple or composite device supporting several USB devices simultaneously.

TinyUSB stack is distributed via IDF Component Registry.

Our USB-OTG implementation is limited to 6 USB endpoints (5 IN/OUT endpoints and 1 IN endpoint). Please note that enabling Secure Boot or flash encryption disables the USB-OTG USB stack in the ROM, disallowing updates via the serial emulation or Device Firmware Update (DFU) on that port. For more details, please refer to technical reference manual.

Features

- Configuration of device and string USB descriptors
- USB Serial Device (CDC-ACM)
- Input and output streams through USB Serial Device
- Other USB classes (MIDI, MSC, HID…) support directly via TinyUSB
- USB Composite Device (MSC + CDC)
- VBUS monitoring for self-powered devices

Hardware USB Connection

- Any board with the ESP32-S2 chip with USB connectors or with exposed USB’s D+ and D- (DATA+/DATA-) pins.

If the board has no USB connector but has the pins, connect pins directly to the host (e.g., with do-it-yourself cable from any USB connection cable).

On ESP32-S2, connect GPIO 20 and 19 to D+/D- respectively:

Self-powered devices must also connect VBUS through voltage divider or comparator, more details in Self-Powered Device subchapter.
Driver Structure

As the basis is used the TinyUSB stack.

On top of it the driver implements:

- Customization of USB descriptors
- Serial device support
- Redirecting of standard streams through the Serial device
- Storage Media (SPI-Flash and SD-Card) for USB Device MSC Class.
- Encapsulated driver’s task servicing the TinyUSB

Configuration

To use the component, you need to add it as a dependency via the following command. For more details, please refer to IDF Component Registry.

```
idf.py add-dependency esp_tinyusb
```

Via Menuconfig options you can specify:

- Several descriptor’s parameters (see Descriptors Configuration below)
- USB Serial low-level configuration
- The verbosity of the TinyUSB’s log
- Disable the TinyUSB main task (for the custom implementation)

Descriptors Configuration

The driver’s descriptors are provided by `tinyusb_config_t` structure’s `device_descriptor`, `configuration_descriptor` and `string_descriptor` members. Therefore, you should initialize `tinyusb_config_t` with your desired descriptors before calling `tinyusb_driver_install()` to install the driver.

However, the driver also provides default descriptors. You can install the driver with default device and string descriptors by setting the `device_descriptor` and `string_descriptor` members of `tinyusb_config_t` to `NULL` before calling `tinyusb_driver_install()`. To lower your development effort we also provide default configuration descriptor for CDC and MSC class, as these classes rarely require custom configuration. The driver’s default device descriptor is specified using Menuconfig, where the following fields should be configured:

- PID
- VID
- bcdDevice
- Manufacturer
- Product name
- Name of CDC or MSC device if it is On
- Serial number

If you want to use your own descriptors with extended modification, you can define them during the driver installation process.

Install Driver

To initialize the driver, users should call `tinyusb_driver_install()`. The driver’s configuration is specified in a `tinyusb_config_t` structure that is passed as an argument to `tinyusb_driver_install()`.

Note that the `tinyusb_config_t` structure can be zero initialized (e.g., `const tinyusb_config_t tusb_cfg = { 0 };`) or partially (as shown below). For any member that is initialized to `0` or `NULL`, the driver will use its default configuration values for that member (see example below).
```c
const tinyusb_config_t partial_init = {
    .deviceDescriptor = NULL,  // Use default device descriptor specified in Menuconfig
    .stringDescriptor = NULL,   // Use default string descriptors specified in Menuconfig
    .externalPhy = false,       // Use internal USB PHY
    .configurationDescriptor = NULL, // Use default configuration descriptor according to settings in Menuconfig
};
```

**Self-Powered Device**

USB specification mandates self-powered devices to monitor voltage level on USB’s VBUS signal. As opposed to bus-powered devices, a self-powered device can be fully functional even without USB connection. The self-powered device detects connection and disconnection events by monitoring the VBUS voltage level. VBUS is considered valid if it rises above 4.75 V and invalid if it falls below 4.35 V.

No ESP32-S2 pin is 5 V tolerant, so you must connect the VBUS to ESP32-S2 via a comparator with voltage thresholds as described above, or use a simple resistor voltage divider that will output (0.75 x Vdd) if VBUS is 4.4 V (see figure below). In both cases, voltage on the sensing pin must be logic low within 3 ms after the device is unplugged from USB host.

![Simple voltage divider for VBUS monitoring](image)

**Fig. 34: Simple voltage divider for VBUS monitoring**

To use this feature, in `tinyusb_config_t` you must set `selfPowered` to `true` and `vbus_monitor_io` to GPIO number that will be used for VBUS monitoring.

**USB Serial Device (CDC-ACM)**

If the CDC option is enabled in Menuconfig, the USB Serial Device can be initialized with `tusb_cdc_acm_init()` according to the settings from `tinyusb_config_cdcacm_t` (see example below).
const tinyusb_config_cdcacm_t acm_cfg = {
    .usb_dev = TINYUSB_USBDEV_0,
    .cdc_port = TINYUSB_CDC_ACM_0,
    .rx_unread_buf_sz = 64,
    .callback_rx = NULL,
    .callback_rx_wanted_char = NULL,
    .callback_line_state_changed = NULL,
    .callback_line_coding_changed = NULL
};
tusb_cdc_acm_init(&acm_cfg);

To specify callbacks you can either set the pointer to your tusb_cdcacm_callback_t function in the configuration structure or call tinyusb_cdcacm_register_callback() after initialization.

USB Serial Console The driver allows to redirect all standard application streams (stdin, stdout, stderr) to the USB Serial Device and return them to UART using esp_tusb_init_console()/esp_tusb_deinit_console() functions.

USB Mass Storage Device (MSC)

If the MSC CONFIG_TINYUSB_MSC_ENABLED option is enabled in Menuconfig, the ESP Chip can be used as USB MSC Device. The storage media (spi-flash or sd-card) can be initialized as shown below (see example below).

- SPI-Flash

```
static esp_err_t storage_init_spiflash(wl_handle_t *wl_handle)
{
    esp_partition_t *data_partition = esp_partition_find_first(ESP_PARTITION_TYPE_DATA, ESP_PARTITION_SUBTYPE_DATA_FAT, NULL);
    wl_mount(data_partition, wl_handle);
}
const tinyusb_msc_spiflash_config_t config_spi = {
    .wl_handle = wl_handle
};
tinyusb_msc_storage_init_spiflash(&config_spi);
```

- SD-Card

```
static esp_err_t storage_init_sdmmc(sdmmc_card_t **card)
{
    sdmmc_host_t host = SDMMC_HOST_DEFAULT();
    sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
    // For SD Card, set bus width to use
    slot_config.width = 4;
    slot_config.clk = CONFIG EXAMPLE_PIN_CLK;
    slot_config.cmd = CONFIG EXAMPLE_PIN_CMD;
    slot_config.d0 = CONFIG EXAMPLE_PIN_D0;
    slot_config.d1 = CONFIG EXAMPLE_PIN_D1;
    slot_config.d2 = CONFIG EXAMPLE_PIN_D2;
    slot_config.d3 = CONFIG EXAMPLE_PIN_D3;
    slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP;
    sd_card = (sdmmc_card_t *)malloc(sizeof(sdmmc_card_t));
```

(continues on next page)
Chapter 2. API Reference

(continued from previous page)

```c
(*host.init)();
sdmmc_host_init_slot(host.slot, (const sdmmc_slot_config_t *) &slot_config);
sdmmc_card_init(&host, sd_card);
```

```c
storage_init_sdmmc(&card);
```

```c
const tinyusb_msc_sdmmc_config_t config_sdmmc = {
.card = card
};
tinyusb_msc_storage_init_sdmmc(&config_sdmmc);
```

**Application Examples**

The table below describes the code examples available in the directory `peripherals/usb/`.

<table>
<thead>
<tr>
<th>Code Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>peripherals/usb/device/tusb_console</code></td>
<td>How to set up ESP32-S2 chip to get log output via Serial Device connection</td>
</tr>
<tr>
<td><code>peripherals/usb/device/tusb_serial_device</code></td>
<td>How to set up ESP32-S2 chip to work as a USB Serial Device</td>
</tr>
<tr>
<td><code>peripherals/usb/device/tusb_midi</code></td>
<td>How to set up ESP32-S2 chip to work as a USB MIDI Device</td>
</tr>
<tr>
<td><code>peripherals/usb/device/tusb_hid</code></td>
<td>How to set up ESP32-S2 chip to work as a USB Human Interface Device</td>
</tr>
<tr>
<td><code>peripherals/usb/device/tusb_msc</code></td>
<td>How to set up ESP32-S2 chip to work as a USB Mass Storage Device</td>
</tr>
<tr>
<td><code>peripherals/usb/device/tusb_composite_msc_serialdevice</code></td>
<td>How to set up ESP32-S2 chip to work as a Composite USB Device (MSC_CDC)</td>
</tr>
</tbody>
</table>

### 2.5.29 USB Host

The document provides information regarding the USB Host Library. This document is split into the following sections:

- **Overview**
- **Architecture**
- **Usage**
- **Examples**
- **Host Stack Configuration**
- **API Reference**
- **Maintainers Notes**

**Overview**

The USB Host Library (hereinafter referred to as the Host Library) is the lowest layer of the USB Host stack that exposes a public facing API. In most cases, applications that require USB Host functionality do not need to interface with the Host Library directly. Instead, most applications use the API provided by a host class driver that is implemented on top of the Host Library.

However, you may want to use the Host Library directly for some of (but not limited to) the following reasons:
- Implementation of a custom host class driver
- Usage of lower level USB Host API

**Features & Limitations**  The Host Library has the following features:

- Supports Full Speed (FS) and Low Speed (LS) Devices.
- Supports all four transfer types, i.e., Control, Bulk, Interrupt, and Isochronous.
- Allows multiple class drivers to run simultaneously, i.e., multiple clients of the Host Library.
- A single device can be used by multiple clients simultaneously, e.g., composite devices.
- The Host Library itself and the underlying Host Stack does not internally instantiate any OS tasks. The number of tasks is entirely controlled by how the Host Library interface is used. However, a general rule of thumb regarding the number of tasks is \((\text{the number of host class drivers running} + 1)\).

Currently, the Host Library and the underlying Host Stack has the following limitations:

- Only supports a single device, but the Host Library’s API is designed for multiple device support.
- Only supports Asynchronous transfers.
- Only supports using the first configuration found. Changing to other configurations is not supported yet.
- Transfer timeouts are not supported yet.

**Architecture**

Fig. 35: Diagram of the key entities involved in USB Host functionality

The diagram above shows the key entities that are involved when implementing USB Host functionality. These entities are:

- **The Host Library**
- **Clients** of the Host Library
- **Devices**
- **Host Library Daemon Task**
Host Library  The Host Library is the lowest public-facing API layer of the ESP-IDF USB Host Stack. Any other ESP-IDF component (such as a class driver or a user component) that needs to communicate with a connected USB device can only do so using the Host Library API either directly or indirectly.

The Host Library’s API is split into two sub-sets, namely the Library API and Client API.

- The Client API handles the communication between a client of the Host Library and one or more USB devices. The Client API should only be called by registered clients of the Host Library.
- The Library API handles all of the Host Library processing that is not specific to a single client, such as device enumeration. Usually, the library API is called by a Host Library Daemon Task.

Clients  A client of the Host Library is a software component, such as a host class driver or user component, which utilizes the Host Library to establish communication with a USB device. Generally, each client has a one-to-one relation with a task. This implies that all Client API calls pertaining to a specific client must originate from the context of the same task.

By organizing the software components that use the Host Library’s into clients, the Host Library can delegate the handling of all events specific to that client to the client’s task. In other words, each client task is responsible for all the required processing and event handling associated with the USB communication that the client initiates.

Daemon Task  Although the Host Library delegates the handling of client events to the clients themselves, there are still Library events –events that are not specific to any particular client—that need to be handled. Library event handling can include things such as:

- Handling USB device connection, enumeration, and disconnection
- Rerouting control transfers to/from clients
- Forwarding events to clients

Therefore, in addition to the client tasks, the Host Library also requires a task, which is usually the Host Library Daemon Task, to handle all of the library events.

Devices  The Host Library shields clients from the details of device handling, encompassing details such as connection, memory allocation, and enumeration. The clients are provided only with a list of already connected and enumerated devices to choose from. During enumeration, each device is automatically configured to use the first configuration found, namely, the first configuration descriptor returned on a Get Configuration Descriptor request.

For most standard devices, the first configuration will have a ConfigurationValue of 1.

It is possible for two or more clients to simultaneously communicate with the same device as long as they are not communicating to the same interface. However, multiple clients can simultaneously communicate with the same device’s default endpoint (i.e., EP0), which will result in their control transfers being serialized.

For a client to communicate with a device, the client must:

1. Open the device using the device’s address. This lets the Host Library know that the client is using that device.
2. Claim the interface(s) that will be used for communication. This prevents other clients from claiming the same interface(s).
3. Send transfers to the endpoints of claimed interfaces. The client’s task is responsible for handling its own processing and events related to USB device communication.

Usage

The Host Library and the underlying Host Stack will not create any tasks. All tasks, namely the client tasks and the Daemon Task, need to be created by the class drivers or the user. Instead, the Host Library provides two event handler functions that handle all of the required Host Library processing, thus these functions should be called repeatedly from the client tasks and the Daemon Task. Therefore, the implementation of client tasks and the Daemon Task will be largely centered around the invocation of these event handler functions.

Host Library & Daemon Task
**Basic Usage**  The Host Library API provides `usb_host_lib_handle_events()` to handle library events. This function should be called repeatedly, typically from the Daemon Task. Some notable features regarding `usb_host_lib_handle_events()` are:

- The function can block until a library event needs handling.
- Event flags are returned on each invocation. These event flags are useful for knowing when the Host Library can be uninstalled.

A bare-bones Daemon Task would resemble something like the following code snippet:

```c
#include "usb/usb_host.h"

void daemon_task(void *arg) {
    ...
    bool exit = false;
    while (!exit) {
        uint32_t event_flags;
        usb_host_lib_handle_events(portMAX_DELAY, &event_flags);
        if (event_flags & USB_HOST_LIB_EVENT_FLAGS_NO_CLIENTS) {
            ...
        }
        if (event_flags & USB_HOST_LIB_EVENT_FLAGS_ALL_FREE) {
            ...
        }
        ...
    }
}
```

**Note:** See the `peripherals/usb/host/usb_host_lib` example for full implementation of the Daemon Task.

![Fig. 36: Graph of Typical USB Host Library Lifecycle](image)

**Lifecycle**  The graph above illustrates the typical lifecycle of the Host Library with multiple clients and devices. Specifically, the example involves:

- two registered clients (Client 1 and Client 2).
- two connected devices (Device 1 and Device 2), where Client 1 communicates with Device 1 and Client 2 communicates with Device 2.

With reference to the graph above, the typical lifecycle involves the following key stages.

1. **The Host Library is installed by calling `usb_host_install()`**.
Chapter 2. API Reference

• Installation must be done before any other Host Library API is called.
• Where `usb_host_install()` is called (e.g., from the Daemon Task or another task) depends on the synchronization logic between the Daemon Task, client tasks, and the rest of the system.

2. **Once the Host Library is installed, the clients can be registered by calling `usb_host_client_register()`**.
   
   • This is typically called from the client task, where the client task waits for a signal from the Daemon Task.
   • This can be called elsewhere if necessary as long as it is called after `usb_host_install()`.

3. **Device 1 connects and is then enumerated.**
   • Each registered client (in this case Client 1 and Client 2) is notified of the new device by way of the `USB_HOST_CLIENT_EVENT_NEW_DEV` event.
   • Client 1 opens Device 1 and begins communication with it.

4. **Similarly Device 2 connects and is enumerated.**
   • Client 1 and 2 are notified of a new device via a `USB_HOST_CLIENT_EVENT_NEW_DEV` event.
   • Client 2 opens Device 2 and begins communication with it.

5. **Device 1 suddenly disconnects.**
   • Client 1 is notified by way of `USB_HOST_CLIENT_EVENT_DEV_GONE` and begins its cleanup.
   • Client 2 is not notified as it has not opened Device 1.

6. **Client 1 completes its cleanup and deregisters by calling `usb_host_client_deregister()`**.
   • This is typically called from the client task before the task exits.
   • This can be called elsewhere if necessary as long as Client 1 has already completed its cleanup.

7. **Client 2 completes its communication with Device 2. Client 2 then closes Device 2 and deregisters itself.**
   • The Daemon Task is notified of the deregistration of all clients by way the `USB_HOST_LIB_EVENT_FLAGS_NO_CLIENTS` event flag as Client 2 is the last client to deregister.
   • Device 2 is still allocated (i.e., not freed), as it is still connected albeit not currently opened by any client.

8. **The Daemon Task decides to clean up as there are no more clients.**
   • The Daemon Task must free Device 2 first by calling `usb_host_device_free_all()`.
   • If `usb_host_device_free_all()` was able to free all devices, the function will return `ESP_OK` indicating that all devices have been freed.
   • If `usb_host_device_free_all()` was unable to free all devices for reasons like the device is still opened by a client, the function will return `ESP_ERR_NOT_FINISHED`.
   • The Daemon Task must wait for `usb_host_lib_handle_events()` to return the `USB_HOST_LIB_EVENT_FLAGS_ALL_FREE` event flag in order to know when all devices have been freed.

9. Once the Daemon Task has verified that all clients have deregistered and all devices have been freed, it can now uninstall the Host Library by calling `usb_host_uninstall()`.

**Clients & Class Driver**

**Basic Usage**  The Host Library API provides `usb_host_client_handle_events()` to handle a particular client’s events. This function should be called repeatedly, typically from the client’s task. Some notable features regarding `usb_host_client_handle_events()` are:

• The function can block until a client event needs handling.
• The function’s primary purpose is to call the various event handling callbacks when a client event occurs.

The following callbacks are called from within `usb_host_client_handle_events()` thus allowing the client task to be notified of events.

• The client event callback of type `usb_host_client_event_cb_t` delivers client event messages to the client. Client event messages indicate events such as the addition or removal of a device.
• The USB transfer completion callback of type `usb_transfer_cb_t` indicates that a particular USB transfer previously submitted by the client has been completed.

**Note:** Given that the callbacks are called from within `usb_host_client_handle_events()`, users should
avoid blocking from within the callbacks as this will result in `usb_host_client_handle_events()` being blocked as well, thus preventing other pending client events from being handled.

The following code snippet demonstrates a bare-bones host class driver and its client task. The code snippet contains:

- A simple client task function `client_task` that calls `usb_host_client_handle_events()` in a loop.
- Implementations of a client event callback and transfer completion callbacks.
- Implementation of a simple state machine for the class driver. The class driver simply opens a device, sends an OUT transfer to EP1, then closes the device.

```c
#include <string.h>
#include "usb/usb_host.h"

#define CLASS_DRIVER_ACTION_OPEN_DEV 0x01
#define CLASS_DRIVER_ACTION_TRANSFER 0x02
#define CLASS_DRIVER_ACTION_CLOSE_DEV 0x03

struct class_driver_control {
    uint32_t actions;
    uint8_t dev_addr;
    usb_host_client_handle_t client_hdl;
    usb_device_handle_t dev_hdl;
};

static void client_event_cb(const usb_host_client_event_msg_t *event_msg, void *arg)
{
    // This function is called from within usb_host_client_handle_events(). Do not block and try to keep it short
    struct class_driver_control *class_driver_obj = (struct class_driver_control *)arg;
    switch (event_msg->event)
    {
        case USB_HOST_CLIENT_EVENT_NEW_DEV:
            class_driver_obj->actions |= CLASS_DRIVER_ACTION_OPEN_DEV;
            class_driver_obj->dev_addr = event_msg->new_dev.address; //Store the address of the new device
            break;
        case USB_HOST_CLIENT_EVENT_DEV_GONE:
            class_driver_obj->actions |= CLASS_DRIVER_ACTION_CLOSE_DEV;
            break;
        default:
            break;
    }
}

static void transfer_cb(usb_transfer_t *transfer)
{
    // This function is called from within usb_host_client_handle_events(). Do not block and try to keep it short
    struct class_driver_control *class_driver_obj = (struct class_driver_control *)transfer->context;
    printf("Transfer status %d, actual number of bytes transferred %d\n", transfer->status, transfer->actual_num_bytes);
    class_driver_obj->actions |= CLASS_DRIVER_ACTION_CLOSE_DEV;
}

void client_task(void *arg)
{
    ... // Wait until Host Library is installed
    // Initialize class driver objects
    struct class_driver_control class_driver_obj = {0};
}
```

(continues on next page)
// Register the client
usb_host_client_config_t client_config = {
    .is_synchronous = false,
    .max_num_event_msg = 5,
    .async = {
        .client_event_callback = client_event_cb,
        .callback_arg = &class_driver_obj,
    }
};
usb_host_client_register(&client_config, &class_driver_obj.client_hdl);

//Allocate a USB transfer
usb_transfer_t *transfer;
usb_host_transfer_alloc(1024, 0, &transfer);

//Event handling loop
bool exit = false;
while (!exit) {
    // Call the client event handler function
    usb_host_client_handle_events(class_driver_obj.client_hdl, portMAX_DELAY);
    // Execute pending class driver actions
    if (class_driver_obj.actions & CLASS_DRIVER_ACTION_OPEN_DEV) {
        // Open the device and claim interface 1
        usb_host_device_open(class_driver_obj.client_hdl, class_driver_obj.dev_
->addr, &class_driver_obj.dev_hdl);
        usb_host_interface_claim(class_driver_obj.client_hdl, class_driver_obj.
->dev_hdl, 1, 0);
        if (class_driver_obj.actions & CLASS_DRIVER_ACTION_TRANSFER) {
            // Send an OUT transfer to EP1
            memset(transfer->data_buffer, 0xAA, 1024);
            transfer->num_bytes = 1024;
            transfer->device_handle = class_driver_obj.dev_hdl;
            transfer->bEndpointAddress = 0x01;
            transfer->callback = transfer_cb;
            transfer->context = (void *)class_driver_obj;
            usb_host_transfer_submit(transfer);
        }
        if (class_driver_obj.actions & CLASS_DRIVER_ACTION_CLOSE_DEV) {
            // Release the interface and close the device
            usb_host_interface_release(class_driver_obj.client_hdl, class_driver_
->obj.dev_hdl, 1);
            usb_host_device_close(class_driver_obj.client_hdl, class_driver_obj.
->dev_hdl);
            exit = true;
        }
        ... // Handle any other actions required by the class driver
    }

    // Cleanup class driver
    usb_host_transfer_free(transfer);
    usb_host_client_deregister(class_driver_obj.client_hdl);
    ... // Delete the client task. Signal the Daemon Task if necessary.
}

Note: An actual host class driver is likely to support many more features, thus will have a much more complex state machine. A host class driver is also likely to need to:

- Be able to open multiple devices
- Parse an opened device’s descriptors to identify if the device is of the target class
- Communicate with multiple endpoints of an interface in a particular order
- Claim multiple interfaces of a device
• Handle various errors

**Lifecycle** The typical life cycle of a client task and class driver will go through the following stages:

1. Wait for some signal regarding the Host Library being installed.
2. Register the client via `usb_host_client_register()` and allocate any other class driver resources, such as allocating transfers using `usb_host_transfer_alloc()`.
3. For each new device that the class driver needs to communicate with:
   a. Check if the device is already connected via `usb_host_device_addr_list_fill()`.
   b. If the device is not already connected, wait for a `USB_HOST_CLIENT_EVENT_NEW_DEV` event from the client event callback.
   c. Open the device via `usb_host_device_open()`.
   d. Parse the device and configuration descriptors via `usb_host_get_device_descriptor()` and `usb_host_get_active_config_descriptor()` respectively.
   e. Claim the necessary interfaces of the device via `usb_host_interface_claim()`.
4. Submit transfers to the device via `usb_host_transfer_submit()` or `usb_host_transfer_submit_control()`.
5. Once an opened device is no longer needed by the class driver, or has disconnected, as indicated by a `USB_HOST_CLIENT_EVENT_DEV_GONE` event:
   a. Stop any previously submitted transfers to the device’s endpoints by calling `usb_host_endpoint_halt()` and `usb_host_endpoint_flush()` on those endpoints.
   b. Release all previously claimed interfaces via `usb_host_interface_release()`.
   c. Close the device via `usb_host_device_close()`.
6. Deregister the client via `usb_host_client_deregister()` and free any other class driver resources.
7. Delete the client task. Signal the Daemon Task if necessary.

**Examples**

**Host Library Examples** The `peripherals/usb/host/usb_host_lib` demonstrates basic usage of the USB Host Library’s API to implement a pseudo-class driver.

**Class Driver Examples** The USB Host Stack provides a number of examples that implement host class drivers using the Host Library’s API.

**CDC-ACM**

- A host class driver for the Communication Device Class (Abstract Control Model) is distributed as a managed component via the ESP-IDF Component Registry.
- The `peripherals/usb/host/cdc/cdc_acm_host` example uses the CDC-ACM host driver component to communicate with CDC-ACM devices.
- The `peripherals/usb/host/cdc/cdc_acm_vcp` example shows how you can extend the CDC-ACM host driver to interface Virtual COM Port devices.
- The CDC-ACM driver is also used in `esp_modem examples`, where it is used for communication with cellular modems.

**MSC**

- A host class driver for the Mass Storage Class (Bulk-Only Transport) is deployed to ESP-IDF Component Registry.
- The `peripherals/usb/host/msc` example demonstrates the usage of the MSC host driver to read and write to a USB flash drive.
HID

- A host class driver for the HID (Human interface device) is distributed as a managed component via the ESP-IDF Component Registry.
- The `peripherals/usb/host/hid` example demonstrates the possibility to receive reports from a USB HID device with several interfaces.

UVC

- A host class driver for the USB Video Device Class is distributed as a managed component via the ESP-IDF Component Registry.
- The `peripherals/usb/host/uvc` example demonstrates the usage of the UVC host driver to receive a video stream from a USB camera and optionally forward that stream over Wi-Fi.

Host Stack Configuration

**Non-Compliant Device Support** To support USB devices that are non-compliant in various scenarios or exhibit specific behaviors, it is possible to configure the USB Host stack.

As a USB device may be hot-plugged, it is essential to have configurable delays between power switching and device attachment, and when the device’s internal power has stabilized.

**Enumeration Configuration** During the process of enumerating connected USB devices, several delay values ensure the proper functioning of the device.

![Fig. 37: USB Root Hub Power-on and Connection Events Timing](image)

The figure above shows all the delay values associated with both turning on port power with a device connected and hot-plugging a device.

- After a port is reset or resumed, the USB system software is expected to provide a “recovery” interval of 10 ms before the device attached to the port is expected to respond to data transfers.
- After the reset/resume recovery interval, if a device receives a `SetAddress()` request, the device must be able to complete processing of the request and be able to successfully complete the Status stage of the request within 50 ms.
- After successful completion of the Status stage, the device is allowed a `SetAddress()` recovery interval of 2 ms.
Note: For more details regarding connection event timings, please refer to USB 2.0 Specification > Chapter 7.1.7.3 Connect and Disconnect Signaling.

Configurable parameters of the USB host stack can be configured with multiple options via Menuconfig.

- For debounce delay, refer to `CONFIG_USB_HOST_DEBOUNCE_DELAY_MS`.
- For reset hold interval, refer to `CONFIG_USB_HOST_RESET_HOLD_MS`.
- For reset recovery interval, refer to `CONFIG_USB_HOST_RESET_RECOVERY_MS`.
- For `SetAddress()` recovery interval, refer to `CONFIG_USB_HOST_SET_ADDR_RECOVERY_MS`.

API Reference

The API of the USB Host Library is separated into the following header files. However, it is sufficient for applications to only `#include "usb/usb_host.h"` and all USB Host Library headers will also be included.

- `usb/include/usb/usb_host.h` contains the functions and types of the USB Host Library.
- `usb/include/usb/usb_helpers.h` contains various helper functions that are related to the USB protocol such as descriptor parsing.
- `usb/include/usb/usb_types_stack.h` contains types that are used across multiple layers of the USB Host stack.
- `usb/include/usb/usb_types_ch9.h` contains types and macros related to Chapter 9 of the USB2.0 specification, i.e., descriptors and standard requests.

Header File

- `components/usb/include/usb/usb_host.h`
  - This header file can be included with:
    ```
    #include "usb/usb_host.h"
    ```
  - This header file is a part of the API provided by the `usb` component. To declare that your component depends on `usb`, add the following to your CMakeLists.txt:
    ```
    REQUIRES usb
    ```
    or
    ```
    PRIV_REQUIRES usb
    ```

Functions

```
esp_err_t usb_host_install (const usb_host_config_t *config)
```
Install the USB Host Library.

- This function should only once to install the USB Host Library
- This function should be called before any other USB Host Library functions are called

Note: If `skip_phy_setup` is set in the install configuration, the user is responsible for ensuring that the underlying Host Controller is enabled and the USB PHY (internal or external) is already setup before this function is called.

Parameters

- `config` - [in] USB Host Library configuration

Returns

- `esp_err_t`
Chapter 2. API Reference

```c
extern esp_err_t usb_host_uninstall(void)

Uninstall the USB Host Library.
```

- This function should be called to uninstall the USB Host Library, thereby freeing its resources
- All clients must have been deregistered before calling this function
- All devices must have been freed by calling `usb_host_device_free_all()` and receiving the `USB.Host.Lib.Event.Flags.ALL_FREE` event flag

**Note:** If `skip_phy_setup` was set when the Host Library was installed, the user is responsible for disabling the underlying Host Controller and USB PHY (internal or external).

```c
extern esp_err_t usb_host_lib_handle_events(TickType_t timeout_ticks, uint32_t* event_flags_ret)

Handle USB Host Library events.
```

- This function handles all of the USB Host Library’s processing and should be called repeatedly in a loop
- Check `event_flags_ret` to see if any flags are set indicating particular USB Host Library events
- This function should never be called by multiple threads simultaneously

**Note:** This function can block

```c
Parameters
  * `timeout_ticks` - [in] Timeout in ticks to wait for an event to occur
  * `event_flags_ret` - [out] Event flags that indicate what USB Host Library event occurred.

Returns esp_err_t
```

```c
extern esp_err_t usb_host_lib_unblock(void)

Unblock the USB Host Library handler.
```

- This function simply unblocks the USB Host Library event handling function (`usb_host_lib_handle_events()`)

```c
Returns esp_err_t
```

```c
extern esp_err_t usb_host_lib_info(usb_host_lib_info_t* info_ret)

Get current information about the USB Host Library.
```

```c
Parameters
  * `info_ret` - [out] USB Host Library Information

Returns esp_err_t
```

```c
extern esp_err_t usb_host_client_register(const usb_host_client_config_t* client_config,
                                          usb_host_client_handle_t* client_hdl_ret)

Register a client of the USB Host Library.
```

- This function registers a client of the USB Host Library
- Once a client is registered, its processing function `usb_host_client_handle_events()` should be called repeatedly

```c
Parameters
```
• **client_config** -[in] Client configuration  
  **client_hdl_ret** -[out] Client handle  

Returns esp_err_t

**esp_err_t usb_host_client_deregister** *(usb_host_client_handle_t client_hdl)*  
Deregister a USB Host Library client.

- This function deregisters a client of the USB Host Library  
- The client must have closed all previously opened devices before attempting to deregister

Parameters  
  **client_hdl** -[in] Client handle  
  Returns esp_err_t

**esp_err_t usb_host_client_handle_events** *(usb_host_client_handle_t client_hdl, TickType_t timeout_ticks)*  
USB Host Library client processing function.

- This function handles all of a client’s processing and should be called repeatedly in a loop  
- For a particular client, this function should never be called by multiple threads simultaneously

**Note:** This function can block

Parameters  
  • **client_hdl** -[in] Client handle  
  • **timeout_ticks** -[in] Timeout in ticks to wait for an event to occur  
  Returns esp_err_t

**esp_err_t usb_host_client_unblock** *(usb_host_client_handle_t client_hdl)*  
Unblock a client.

- This function simply unblocks a client if it is blocked on the usb_host_client_handle_events() function.  
- This function is useful when need to unblock a client in order to deregister it.

Parameters  
  **client_hdl** -[in] Client handle  
  Returns esp_err_t

**esp_err_t usb_host_device_open** *(usb_host_client_handle_t client_hdl, uint8_t dev_addr, usb_device_handle_t *dev_hdl_ret)*  
Open a device.

- This function allows a client to open a device  
- A client must open a device first before attempting to use it (e.g., sending transfers, device requests etc.)

Parameters  
  • **client_hdl** -[in] Client handle  
  • **dev_addr** -[in] Device’s address  
  • **dev_hdl_ret** -[out] Device’s handle  
  Returns esp_err_t
**esp_err_t usb_host_device_close** *(usb_host_client_handle_t client_hdl, usb_device_handle_t dev_hdl)*

Close a device.

- This function allows a client to close a device
- A client must close a device after it has finished using the device (claimed interfaces must also be released)
- A client must close all devices it has opened before deregistering

**Note:** This function can block

**Parameters**

- **client_hdl** -[in] Client handle
- **dev_hdl** -[in] Device handle

**Returns** esp_err_t

**esp_err_t usb_host_device_free_all** *(void)*

Indicate that all devices can be freed when possible.

- This function marks all devices as waiting to be freed
- If a device is not opened by any clients, it will be freed immediately
- If a device is opened by at least one client, the device will be free when the last client closes that device.
- Wait for the USB_HOST_LIB_EVENT_FLAGS_ALL_FREE flag to be set by usb_host_lib_handle_events() in order to know when all devices have been freed
- This function is useful when cleaning up devices before uninstalling the USB Host Library

**Returns**

- ESP_ERR_NOT_FINISHED: There are one or more devices that still need to be freed. Wait for USB_HOST_LIB_EVENT_FLAGS_ALL_FREE event
- ESP_OK: All devices already freed (i.e., there were no devices)
- Other: Error

**esp_err_t usb_host_device_addr_list_fill** *(int list_len, uint8_t *dev_addr_list, int *num_dev_ret)*

Fill a list of device address.

- This function fills an empty list with the address of connected devices
- The Device addresses can then used in usb_host_device_open()
- If there are more devices than the list_len, this function will only fill up to list_len number of devices.

**Parameters**

- **list_len** -[in] Length of the empty list
- **dev_addr_list** -[inout] Empty list to be filled
- **num_dev_ret** -[out] Number of devices

**Returns** esp_err_t

**esp_err_t usb_host_device_info** *(usb_device_handle_t dev_hdl, usb_device_info_t *dev_info)*

Get device’s information.

- This function gets some basic information of a device
- The device must be opened first before attempting to get its information

**Note:** This function can block
Parameters

- `dev_hdl` - [in] Device handle
- `dev_info` - [out] Device information

Returns `esp_err_t`

`esp_err_t usb_host_get_device_descriptor`(`usb_device_handle_t` dev_hdl, `const` `usb_device_desc_t` **device_desc)

Get device’s device descriptor.

- A client must call `usb_host_device_open()` first
- No control transfer is sent. The device’s descriptor is cached on enumeration
- This function simple returns a pointer to the cached descriptor

Note: No control transfer is sent. The device’s descriptor is cached on enumeration

Parameters

- `dev_hdl` - [in] Device handle
- `device_desc` - [out] Device descriptor

Returns `esp_err_t`

`esp_err_t usb_host_get_active_config_descriptor`(`usb_device_handle_t` dev_hdl, `const` `usb_config_desc_t` **config_desc)

Get device’s active configuration descriptor.

- A client must call `usb_host_device_open()` first
- No control transfer is sent. The device’s active configuration descriptor is cached on enumeration
- This function simple returns a pointer to the cached descriptor

Note: This function can block

Note: No control transfer is sent. A device’s active configuration descriptor is cached on enumeration

Parameters

- `dev_hdl` - [in] Device handle
- `config_desc` - [out] Configuration descriptor

Returns `esp_err_t`

`esp_err_t usb_host_interface_claim`(`usb_host_client_handle_t` client_hdl, `usb_device_handle_t` dev_hdl, `uint8_t` bInterfaceNumber, `uint8_t` bAlternateSetting)

Function for a client to claim a device’s interface.

- A client must claim a device’s interface before attempting to communicate with any of its endpoints
- Once an interface is claimed by a client, it cannot be claimed by any other client.

Note: This function can block

Parameters

- `client_hdl` - [in] Client handle
dev_hdl – [in] Device handle
bInterfaceNumber – [in] Interface number
bAlternateSetting – [in] Interface alternate setting number

Returns esp_err_t

esp_err_t usb_host_interface_release (usb_host_client_handle_t client_hdl, usb_device_handle_t dev_hdl, uint8_t bInterfaceNumber)

Function for a client to release a previously claimed interface.

• A client should release a device’s interface after it no longer needs to communicate with the interface
• A client must release all of its interfaces of a device it has claimed before being able to close the device

Note: This function can block

Parameters
• client_hdl – [in] Client handle
• dev_hdl – [in] Device handle
• bInterfaceNumber – [in] Interface number

Returns esp_err_t

esp_err_t usb_host_endpoint_halt (usb_device_handle_t dev_hdl, uint8_t bEndpointAddress)

Halt a particular endpoint.

• The device must have been opened by a client
• The endpoint must be part of an interface claimed by a client
• Once halted, the endpoint must be cleared using usb_host_endpoint_clear() before it can communicate again

Note: This function can block

Parameters
• dev_hdl – Device handle
• bEndpointAddress – Endpoint address

Returns esp_err_t

esp_err_t usb_host_endpoint_flush (usb_device_handle_t dev_hdl, uint8_t bEndpointAddress)

Flush a particular endpoint.

• The device must have been opened by a client
• The endpoint must be part of an interface claimed by a client
• The endpoint must have been halted (either through a transfer error, or usb_host_endpoint_halt())
• Flushing an endpoint will caused an queued up transfers to be canceled

Note: This function can block

Parameters
• dev_hdl – Device handle
• bEndpointAddress – Endpoint address

Returns esp_err_t
**esp_err_t usb_host_endpoint_clear (usb_device_handle_t dev_hdl, uint8_t bEndpointAddress)**

Clear a halt on a particular endpoint.

- The device must have been opened by a client
- The endpoint must be part of an interface claimed by a client
- The endpoint must have been halted (either through a transfer error, or `usb_host_endpoint_halt()`)
- If the endpoint has any queued up transfers, clearing a halt will resume their execution

**Note:** This function can block

**Parameters**
- `dev_hdl` – Device handle
- `bEndpointAddress` – Endpoint address

**Returns** `esp_err_t`

**esp_err_t usb_host_transfer_alloc (size_t data_buffer_size, int num_isoc_packets, usb_transfer_t **transfer)**

Allocate a transfer object.

- This function allocates a transfer object
- Each transfer object has a fixed sized buffer specified on allocation
- A transfer object can be re-used indefinitely
- A transfer can be submitted using `usb_host_transfer_submit()` or `usb_host_transfer_submit_control()`

**Parameters**
- `data_buffer_size` – [in] Size of the transfer’s data buffer
- `num_isoc_packets` – [in] Number of isochronous packets in transfer (set to 0 for non-isochronous transfers)
- `transfer` – [out] Transfer object

**Returns** `esp_err_t`

**esp_err_t usb_host_transfer_free (usb_transfer_t *transfer)**

Free a transfer object.

- Free a transfer object previously allocated using `usb_host_transfer_alloc()`
- The transfer must not be in-flight when attempting to free it
- If a NULL pointer is passed, this function will simply return ESP_OK

**Parameters** `transfer` – [in] Transfer object

**Returns** `esp_err_t`

**esp_err_t usb_host_transfer_submit (usb_transfer_t *transfer)**

Submit a non-control transfer.

- Submit a transfer to a particular endpoint. The device and endpoint number is specified inside the transfer
- The transfer must be properly initialized before submitting
- On completion, the transfer’s callback will be called from the client’s `usb_host_client_handle_events()` function.

**Parameters** `transfer` – [in] Initialized transfer object

**Returns** `esp_err_t`
esp_err_t usb_host_transfer_submit_control(usb_host_client_handle_t client_hdl, usb_transfer_t *transfer)

Submit a control transfer.

- Submit a control transfer to a particular device. The client must have opened the device first
- The transfer must be properly initialized before submitting. The first 8 bytes of the transfer’s data buffer should contain the control transfer setup packet
- On completion, the transfer’s callback will be called from the client’s usb_host_client_handle_events() function.

Parameters
- client_hdl [in] Client handle
- transfer [in] Initialized transfer object

Returns esp_err_t

Structures

struct usb_host_client_event_msg_t

Client event message.

Client event messages are sent to each client of the USB Host Library in order to notify them of various USB Host Library events such as:

- Addition of new devices
- Removal of existing devices

Note: The event message structure has a union with members corresponding to each particular event. Based on the event type, only the relevant member field should be accessed.

Public Members

usb_host_client_event_t event
Type of event

uint8_t address
New device’s address

struct usb_host_client_event_msg_t::[anonymous]::[anonymous] new_dev
New device info

usb_device_handle_t dev_hdl
The handle of the device that was gone

struct usb_host_client_event_msg_t::[anonymous]::[anonymous] dev_gone
Gone device info

struct usb_host_lib_info_t
Current information about the USB Host Library obtained via usb_host_lib_info()
Public Members

`int num_devices`
Current number of connected (and enumerated) devices

`int num_clients`
Current number of registered clients

`struct usb_host_config_t`
USB Host Library configuration.
Configuration structure of the USB Host Library. Provided in the `usb_host_install()` function

Public Members

`bool skip_phy_setup`
If set, the USB Host Library will not configure the USB PHY thus allowing the user to manually configure the USB PHY before calling `usb_host_install()`. Users should set this if they want to use an external USB PHY. Otherwise, the USB Host Library will automatically configure the internal USB PHY

`int intr_flags`
Interrupt flags for the underlying ISR used by the USB Host stack

`struct usb_host_client_config_t`
USB Host Library Client configuration.
Configuration structure for a USB Host Library client. Provided in `usb_host_client_register()`

Public Members

`bool is_synchronous`
Whether the client is asynchronous or synchronous or not. Set to false for now.

`int max_num_event_msg`
Maximum number of event messages that can be stored (e.g., 3)

`usb_host_client_event_cb_t client_event_callback`
Client’s event callback function

`void *callback_arg`
Event callback function argument

`struct usb_host_client_config_t::[anonymous]::[anonymous] async`
Async callback config

Macros

`USB_HOST_LIB_EVENT_FLAGS_NO_CLIENTS`
All clients have been deregistered from the USB Host Library
USB_HOST_LIB_EVENT_FLAGS_ALL_FREE

The USB Host Library has freed all devices

Type Definitions

typedef struct usb_host_client_handle_s *usb_host_client_handle_t

Handle to a USB Host Library asynchronous client.

An asynchronous client can be registered using usb_host_client_register()

Note: Asynchronous API

typedef void (*usb_host_client_event_cb_t)(const usb_host_client_event_msg_t *event_msg, void *arg)

Client event callback.

• Each client of the USB Host Library must register an event callback to receive event messages from the USB Host Library.
• The client event callback is run from the context of the clients usb_host_client_handle_events() function

Enumerations

enum usb_host_client_event_t

The type event in a client event message.

Values:

enumerator USB_HOST_CLIENT_EVENT_NEW_DEV

A new device has been enumerated and added to the USB Host Library

enumerator USB_HOST_CLIENT_EVENT_DEV_GONE

A device opened by the client is now gone

Header File

• components/usb/include/usb/usb_helpers.h

• This header file can be included with:

```
#include "usb/usb_helpers.h"
```

• This header file is a part of the API provided by the usb component. To declare that your component depends on usb, add the following to your CMakeLists.txt:

```
REQUIRES usb
```

or

```
PRIV_REQUIRES usb
```

Functions

cnst usb_standard_desc_t *usb_parse_next_descriptor (const usb_standard_desc_t *cur_desc, uint16_t wTotalLength, int *offset)
Get the next descriptor.

Given a particular descriptor within a full configuration descriptor, get the next descriptor within the configuration descriptor. This is a convenience function that can be used to walk each individual descriptor within a full configuration descriptor.

**Parameters**
- `cur_desc` [in] Current descriptor
- `wTotalLength` [in] Total length of the configuration descriptor
- `offset` [inout] Byte offset relative to the start of the configuration descriptor. On input, it is the offset of the current descriptor. On output, it is the offset of the returned descriptor.

**Returns** `usb_standard_desc_t*` Next descriptor, NULL if end of configuration descriptor reached

```
const usb_standard_desc_t* usb_parse_next_descriptor_of_type(const usb_standard_desc_t* cur_desc, uint16_t wTotalLength, uint8_t bDescriptorType, int* offset)
```

Get the next descriptor of a particular type.

Given a particular descriptor within a full configuration descriptor, get the next descriptor of a particular type (i.e., using the `bDescriptorType` value) within the configuration descriptor.

**Parameters**
- `cur_desc` [in] Current descriptor
- `wTotalLength` [in] Total length of the configuration descriptor
- `bDescriptorType` [in] Type of the next descriptor to get
- `offset` [inout] Byte offset relative to the start of the configuration descriptor. On input, it is the offset of the current descriptor. On output, it is the offset of the returned descriptor.

**Returns** `usb_standard_desc_t*` Next descriptor, NULL if end descriptor is not found or configuration descriptor reached

```
int usb_parse_interface_number_of_alternate(const usb_config_desc_t* config_desc, uint8_t bInterfaceNumber)
```

Get the number of alternate settings for a `bInterfaceNumber`.

Given a particular configuration descriptor, for a particular `bInterfaceNumber`, get the number of alternate settings available for that interface (i.e., the max possible value of `bAlternateSetting` for that `bInterfaceNumber`).

**Parameters**
- `config_desc` [in] Pointer to the start of a full configuration descriptor
- `bInterfaceNumber` [in] Interface number

**Returns** `int` The number of alternate settings that the interface has, -1 if `bInterfaceNumber` not found

```
const usb_intf_desc_t* usb_parse_interface_descriptor(const usb_config_desc_t* config_desc, uint8_t bInterfaceNumber, uint8_t bAlternateSetting, int* offset)
```

Get a particular interface descriptor (using `bInterfaceNumber` and `bAlternateSetting`)

Given a full configuration descriptor, get a particular interface descriptor.

---

**Note:** To get the number of alternate settings for a particular `bInterfaceNumber`, call `usb_parse_interface_number_of_alternate()`.

**Parameters**
- `config_desc` [in] Pointer to the start of a full configuration descriptor
- `bInterfaceNumber` [in] Interface number
- `bAlternateSetting` [in] Alternate setting number
• offset - [out] Byte offset of the interface descriptor relative to the start of the configuration descriptor. Can be NULL.

**Returns** const usb_intf_desc_t* Pointer to interface descriptor, NULL if not found.

```c
const usb_ep_desc_t *usb_parse_endpoint_descriptor_by_index(const usb_intf_desc_t *intf_desc,
int index, uint16_t wTotalLength,
int *offset)
```

Get an endpoint descriptor within an interface descriptor.

Given an interface descriptor, get the Nth endpoint descriptor of the interface. The number of endpoints in an interface is indicated by the bNumEndpoints field of the interface descriptor.

**Note:** If bNumEndpoints is 0, it means the interface uses the default endpoint only

**Parameters**
- intf_desc - [in] Pointer to the start of an interface descriptor
- index - [in] Endpoint index
- wTotalLength - [in] Total length of the containing configuration descriptor
- offset - [in/out] Byte offset relative to the start of the configuration descriptor. On input, it is the offset of the interface descriptor. On output, it is the offset of the endpoint descriptor.

**Returns** const usb_ep_desc_t* Pointer to endpoint descriptor, NULL if not found.

```c
const usb_ep_desc_t *usb_parse_endpoint_descriptor_by_address(const usb_config_desc_t *config_desc,uint8_t bInterfaceNumber,uint8_t bAlternateSetting,uint8_t bEndpointAddress,int *offset)
```

Get an endpoint descriptor based on an endpoint’s address.

Given a configuration descriptor, get an endpoint descriptor based on its bEndpointAddress, bAlternateSetting, and bInterfaceNumber.

**Parameters**
- config_desc - [in] Pointer to the start of a full configuration descriptor
- bInterfaceNumber - [in] Interface number
- bAlternateSetting - [in] Alternate setting number
- bEndpointAddress - [in] Endpoint address
- offset - [out] Byte offset of the endpoint descriptor relative to the start of the configuration descriptor. Can be NULL

**Returns** const usb_ep_desc_t* Pointer to endpoint descriptor, NULL if not found.

```c
void usb_print_device_descriptor (const usb_device_desc_t *devc_desc)
```

Print device descriptor.

**Parameters** devc_desc - Device descriptor

```c
void usb_print_config_descriptor (const usb_config_desc_t *cfg_desc, print_class_descriptor_cb class_specific_cb)
```

Print configuration descriptor.

- This function prints the full contents of a configuration descriptor (including interface and endpoint descriptors)
- When a non-standard descriptor is encountered, this function will call the class_specific_cb if it is provided

**Parameters**
void *usb_print_string_descriptor(const *str_desc)

Print a string descriptor.

This function will only print ASCII characters of the UTF-16 encoded string.

Parameters

str_desc — String descriptor

static inline int usb_round_up_to_mps(int num_bytes, int mps)

Round up to an integer multiple of endpoint’s MPS.

This is a convenience function to round up a size/length to an endpoint’s MPS (Maximum packet size). This is useful when calculating transfer or buffer lengths of IN endpoints.

- If MPS <= 0, this function will return 0
- If num_bytes <= 0, this function will return 0

Parameters

num_bytes — [in] Number of bytes
mps — [in] MPS

Returns

int Round up integer multiple of MPS

Type Definitions

typedef void (*print_class_descriptor_cb)(const *usb_standard_desc_t)

Print class specific descriptor callback.

Optional callback to be provided to usb_print_config_descriptor() function. The callback is called when a non-standard descriptor is encountered. The callback should decode the descriptor as print it.

Header File

- components/usb/include/usb/usb_types_stack.h
- This header file can be included with:

```c
#include "usb/usb_types_stack.h"
```

- This header file is a part of the API provided by the usb component. To declare that your component depends on usb, add the following to your CMakeLists.txt:

```c
REQUIRES usb
```

or

```c
PRIV_REQUIRES usb
```

Structures

struct usb_device_info_t

Basic information of an enumerated device.

Public Members


```c
usb_speed_t speed
```

Device’s speed
Chapter 2  API Reference

```c
uint8_t dev_addr
    Device’s address

uint8_t bMaxPacketSize0
    The maximum packet size of the device’s default endpoint

uint8_t bConfigurationValue
    Device’s current configuration number

const usb_str_desc_t *str_desc_manufacturer
    Pointer to Manufacturer string descriptor (can be NULL)

const usb_str_desc_t *str_desc_product
    Pointer to Product string descriptor (can be NULL)

const usb_str_desc_t *str_desc_serial_num
    Pointer to Serial Number string descriptor (can be NULL)

struct usb_isoc_packet_desc_t
    Isochronous packet descriptor.

    If the number of bytes in an Isochronous transfer is larger than the MPS of the endpoint, the transfer is split into multiple packets transmitted at the endpoint’s specified interval. An array of Isochronous packet descriptors describes how an Isochronous transfer should be split into multiple packets.

Public Members

int num_bytes
    Number of bytes to transmit/receive in the packet. IN packets should be integer multiple of MPS

int actual_num_bytes
    Actual number of bytes transmitted/received in the packet

usb_transfer_status_t status
    Status of the packet

struct usb_transfer_s
    USB transfer structure.

Public Members

uint8_t *const data_buffer
    Pointer to data buffer

const size_t data_buffer_size
    Size of the data buffer in bytes
```
int num_bytes
   Number of bytes to transfer. Control transfers should include the size of the setup packet. Isochronous transfer should be the total transfer size of all packets. For non-control IN transfers, num_bytes should be an integer multiple of MPS.

int actual_num_bytes
   Actual number of bytes transferred

uint32_t flags
   Transfer flags

usb_device_handle_t device_handle
   Device handle

uint8_t bEndpointAddress
   Endpoint Address

usb_transfer_status_t status
   Status of the transfer

uint32_t timeout_ms
   Timeout (in milliseconds) of the packet (currently not supported yet)

usb_transfer_cb_t callback
   Transfer callback

void *context
   Context variable for transfer to associate transfer with something

const int num_isoc_packets
   Only relevant to Isochronous. Number of service periods (i.e., intervals) to transfer data buffer over.

usb_isoc_packet_desc_t isoc_packet_desc[]
   Descriptors for each Isochronous packet

Macros

USB_TRANSFER_FLAG_ZERO_PACK
   Terminate Bulk/Interrupt OUT transfer with a zero length packet.

OUT transfers normally terminate when the Host has transferred the exact amount of data it needs to the device. However, for bulk and interrupt OUT transfers, if the transfer size just happened to be a multiple of MPS, it will be impossible to know the boundary between two consecutive transfers to the same endpoint. Therefore, this flag will cause the transfer to automatically add a zero length packet (ZLP) at the end of the transfer if the following conditions are met:
   • The target endpoint is a Bulk/Interrupt OUT endpoint (Host to device)
   • The transfer’s length (i.e., transfer.num_bytes) is a multiple of the endpoint’s MPS

Otherwise, this flag has no effect.

Users should check whether their target device’s class requires a ZLP, as not all Bulk/Interrupt OUT endpoints require them. For example:
• For MSC Bulk Only Transport class, the Host MUST NEVER send a ZLP. Bulk transfer boundaries are determined by the CBW and CSW instead
• For CDC Ethernet, the Host MUST ALWAYS send a ZLP if a segment (i.e., a transfer) is a multiple of MPS (See 3.3.1 Segment Delineation)

**Note:** See USB2.0 specification 5.7.3 and 5.8.3 for more details

**Note:** IN transfers normally terminate when the Host as receive the exact amount of data it needs (must be multiple of MPS) or the endpoint sends a short packet to the Host (For bulk OUT only). Indicates that a bulk OUT transfers should always terminate with a short packet, even if it means adding an extra zero length packet

### Type Definitions

typedef struct usb_device_handle_s *usb_device_handle_t
Handle of a USB Device connected to a USB Host.

typedef struct usb_transfer_s usb_transfer_t
USB transfer structure.

This structure is used to represent a transfer from a software client to an endpoint over the USB bus. Some of the fields are made const on purpose as they are fixed on allocation. Users should call the appropriate USB Host Library function to allocate a USB transfer structure instead of allocating this structure themselves.

The transfer type is inferred from the endpoint this transfer is sent to. Depending on the transfer type, users should note the following:

• Bulk: This structure represents a single bulk transfer. If the number of bytes exceeds the endpoint’s MPS, the transfer will be split into multiple MPS sized packets followed by a short packet.
• Control: This structure represents a single control transfer. This first 8 bytes of the data_buffer must be filled with the setup packet (see `usb_setup_packet_t`). The num_bytes field should be the total size of the transfer (i.e., size of setup packet + wLength).
• Interrupt: Represents an interrupt transfer. If num_bytes exceeds the MPS of the endpoint, the transfer will be split into multiple packets, and each packet is transferred at the endpoint’s specified interval.
• Isochronous: Represents a stream of bytes that should be transferred to an endpoint at a fixed rate. The transfer is split into packets according to the each isoc_packet_desc. A packet is transferred at each interval of the endpoint. If an entire ISOC URB was transferred without error (skipped packets do not count as errors), the URB’s overall status and the status of each packet descriptor will be updated, and the actual_num_bytes reflects the total bytes transferred over all packets. If the ISOC URB encounters an error, the entire URB is considered erroneous so only the overall status will updated.

**Note:** For Bulk/Control/Interrupt IN transfers, the num_bytes must be an integer multiple of the endpoint’s MPS

**Note:** This structure should be allocated via `usb_host_transfer_alloc()`

**Note:** Once the transfer has be submitted, users should not modify the structure until the transfer has completed

typedef void (*usb_transfer_cb_t)(usb_transfer_t *transfer)
USB transfer completion callback.

**Enumerations**

enum *usb_speed_t*

USB Standard Speeds.

*Values:*

- enumerator **USB_SPEED_LOW**
  USB Low Speed (1.5 Mbit/s)

- enumerator **USB_SPEED_FULL**
  USB Full Speed (12 Mbit/s)

enum *usb_transfer_type_t*

The type of USB transfer.

*Note:* The enum values need to match the bmAttributes field of an EP descriptor

*Values:*

- enumerator **USB_TRANSFER_TYPE_CTRL**

- enumerator **USB_TRANSFER_TYPE_ISOCHRONOUS**

- enumerator **USB_TRANSFER_TYPE_BULK**

- enumerator **USB_TRANSFER_TYPE_INTR**

enum *usb_transfer_status_t*

The status of a particular transfer.

*Values:*

- enumerator **USB_TRANSFER_STATUS_COMPLETED**
  The transfer was successful (but may be short)

- enumerator **USB_TRANSFER_STATUS_ERROR**
  The transfer failed because due to excessive errors (e.g. no response or CRC error)

- enumerator **USB_TRANSFER_STATUS_TIMED_OUT**
  The transfer failed due to a time out

- enumerator **USB_TRANSFER_STATUS_CANCELED**
  The transfer was canceled

- enumerator **USB_TRANSFER_STATUSSTALL**
  The transfer was stalled
enumerator **USB_TRANSFER_STATUS_OVERFLOW**

The transfer as more data was sent than was requested

enumerator **USB_TRANSFER_STATUS_SKIPPED**

ISOC packets only. The packet was skipped due to system latency or bus overload

enumerator **USB_TRANSFER_STATUS_NO_DEVICE**

The transfer failed because the target device is gone

**Header File**

- components/usb/include/usb/usb_types_ch9.h
- This header file can be included with:

```
#include "usb/usb_types_ch9.h"
```

- This header file is a part of the API provided by the `usb` component. To declare that your component depends on `usb`, add the following to your CMakeLists.txt:

```
REQUIRES usb
```

or

```
PRIV_REQUIRES usb
```

**Unions**

union **usb_setup_packet_t**

```c
#include <usb_types_ch9.h>
```

Structure representing a USB control transfer setup packet.

See Table 9-2 of USB2.0 specification for more details

**Public Members**

- **uint8_t bmRequestType**
  - Characteristics of request

- **uint8_t bRequest**
  - Specific request

- **uint16_t wValue**
  - Word-sized field that varies according to request

- **uint16_t wIndex**
  - Word-sized field that varies according to request; typically used to pass an index or offset

- **uint16_t wLength**
  - Number of bytes to transfer if there is a data stage

struct **usb_setup_packet_t::[anonymous]**

[anonymous]
Chapter 2. API Reference

```c
uint8_t val[USB_SETUP_PACKET_SIZE]
    Descriptor value

union usb_standard_desc_t
    #include <usb_types_ch9.h> USB standard descriptor.
    All USB standard descriptors start with these two bytes. Use this type when traversing over configuration descriptors

    Public Members

    uint8_t bLength
        Size of the descriptor in bytes

    uint8_t bDescriptorType
        Descriptor Type

    struct usb_standard_desc_t: :[anonymous] USB_DESC_ATTR
        USB descriptor attributes

    uint8_t val[USB_STANDARD_DESC_SIZE]
        Descriptor value

union usb_device_desc_t
    #include <usb_types_ch9.h> Structure representing a USB device descriptor.
    See Table 9-8 of USB2.0 specification for more details

    Public Members

    uint8_t bLength
        Size of the descriptor in bytes

    uint8_t bDescriptorType
        DEVICE Descriptor Type

    uint16_t bcdUSB
        USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H)

    uint8_t bDeviceClass
        Class code (assigned by the USB-IF)

    uint8_t bDeviceSubClass
        Subclass code (assigned by the USB-IF)

    uint8_t bDeviceProtocol
        Protocol code (assigned by the USB-IF)
```
Chapter 2. API Reference

```c
uint8_t bMaxPacketSize0
    Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)

uint16_t idVendor
    Vendor ID (assigned by the USB-IF)

uint16_t idProduct
    Product ID (assigned by the manufacturer)

uint16_t bcdDevice
    Device release number in binary-coded decimal

uint8_t iManufacturer
    Index of string descriptor describing manufacturer

uint8_t iProduct
    Index of string descriptor describing product

uint8_t iSerialNumber
    Index of string descriptor describing the device’s serial number

uint8_t bNumConfigurations
    Number of possible configurations

struct usb_device_desc_t::[anonymous] USB_DESC_ATTR
    USB descriptor attributes

uint8_t val[USB_DEVICE_DESC_SIZE]
    Descriptor value

union usb_config_desc_t

#include <usb_types_ch9.h> Structure representing a short USB configuration descriptor.

See Table 9-10 of USB2.0 specification for more details
```

**Note:** The full USB configuration includes all the interface and endpoint descriptors of that configuration.

**Public Members**

```c
uint8_t bLength
    Size of the descriptor in bytes

uint8_t bDescriptorType
    CONFIGURATION Descriptor Type

uint16_t wTotalLength
    Total length of data returned for this configuration
```
uint8_t bNumInterfaces
    Number of interfaces supported by this configuration

uint8_t bConfigurationValue
    Value to use as an argument to the SetConfiguration() request to select this configuration

uint8_t iConfiguration
    Index of string descriptor describing this configuration

uint8_t bmAttributes
    Configuration characteristics

uint8_t bMaxPower
    Maximum power consumption of the USB device from the bus in this specific configuration when the
device is fully operational.

struct usb_config_desc_t::[anonymous] USB_DESC_ATTR
    USB descriptor attributes

uint8_t val[USB_CONFIG_DESC_SIZE]
    Descriptor value

union usb_iad_desc_t
    #include <usb_types_ch9.h> Structure representing a USB interface association descriptor.

Public Members

uint8_t bLength
    Size of the descriptor in bytes

uint8_t bDescriptorType
    INTERFACE ASSOCIATION Descriptor Type

uint8_t bFirstInterface
    Interface number of the first interface that is associated with this function

uint8_t bInterfaceCount
    Number of contiguous interfaces that are associated with this function

uint8_t bFunctionClass
    Class code (assigned by USB-IF)

uint8_t bFunctionSubClass
    Subclass code (assigned by USB-IF)

uint8_t bFunctionProtocol
    Protocol code (assigned by USB-IF)


```c
uint8_t iFunction
  Index of string descriptor describing this function

struct usb_iad_desc_t::[anonymous] USB_DESC_ATTR
  USB descriptor attributes

uint8_t val[USB_IAD_DESC_SIZE]
  Descriptor value

union usb_intf_desc_t
  #include <usb_types_ch9.h> Structure representing a USB interface descriptor.
  See Table 9-12 of USB2.0 specification for more details

Public Members

uint8_t bLength
  Size of the descriptor in bytes

uint8_t bDescriptorType
  INTERFACE Descriptor Type

uint8_t bInterfaceNumber
  Number of this interface.

uint8_t bAlternateSetting
  Value used to select this alternate setting for the interface identified in the prior field

uint8_t bNumEndpoints
  Number of endpoints used by this interface (excluding endpoint zero).

uint8_t bInterfaceClass
  Class code (assigned by the USB-IF)

uint8_t bInterfaceSubClass
  Subclass code (assigned by the USB-IF)

uint8_t bInterfaceProtocol
  Protocol code (assigned by the USB)

uint8_t iInterface
  Index of string descriptor describing this interface

struct usb_intf_desc_t::[anonymous] USB_DESC_ATTR
  USB descriptor attributes

uint8_t val[USB_INTF_DESC_SIZE]
  Descriptor value
```
union **usb_ep_desc_t**  
#include <usb_types_ch9.h> Structure representing a USB endpoint descriptor.  
See Table 9-13 of USB2.0 specification for more details

**Public Members**

- **uint8_t** `bLength`  
  Size of the descriptor in bytes
- **uint8_t** `bDescriptorType`  
  ENDPOINT Descriptor Type
- **uint8_t** `bEndpointAddress`  
  The address of the endpoint on the USB device described by this descriptor
- **uint8_t** `bmAttributes`  
  This field describes the endpoint’s attributes when it is configured using the `bConfigurationValue`.
- **uint16_t** `wMaxPacketSize`  
  Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected.
- **uint8_t** `bInterval`  
  Interval for polling Isochronous and Interrupt endpoints. Expressed in frames or microframes depending on the device operating speed (1 ms for Low-Speed and Full-Speed or 125 us for USB High-Speed and above).

struct **usb_ep_desc_t::[anonymous]**  
USB descriptor attributes

- **uint8_t** `val[USB_EP_DESC_SIZE]`  
  Descriptor value

union **usb_str_desc_t**  
#include <usb_types_ch9.h> Structure representing a USB string descriptor.

**Public Members**

- **uint8_t** `bLength`  
  Size of the descriptor in bytes
- **uint8_t** `bDescriptorType`  
  STRING Descriptor Type
- **uint16_t** `wData[]`  
  UTF-16LE encoded
struct usb_str_desc_t::[anonymous] USB_DESC_ATTR
    USB descriptor attributes

    uint8_t val[USB_STR_DESC_SIZE]
    Descriptor value

Macros

USB_DESC_ATTR

USB_B_DESCRIPTOR_TYPE_DEVICE
    Descriptor types from USB2.0 specification table 9.5.

USB_B_DESCRIPTOR_TYPE_CONFIGURATION

USB_B_DESCRIPTOR_TYPE_STRING

USB_B_DESCRIPTOR_TYPE_INTERFACE

USB_B_DESCRIPTOR_TYPE_ENDPOINT

USB_B_DESCRIPTOR_TYPE_DEVICE_QUALIFIER

USB_B_DESCRIPTOR_TYPE_OTHER_SPEED_CONFIGURATION

USB_B_DESCRIPTOR_TYPE_INTERFACE_POWER

USB_B_DESCRIPTOR_TYPE_SOF
    Descriptor types from USB 2.0 ECN.

USB_B_DESCRIPTOR_TYPE_DEBUG

USB_B_DESCRIPTOR_TYPE_INTERFACE_ASSOCIATION

USB_B_DESCRIPTOR_TYPE_SECURITY
    Descriptor types from Wireless USB spec.

USB_B_DESCRIPTOR_TYPE_KEY

USB_B_DESCRIPTOR_TYPE_ENCRYPTION_TYPE

USB_B_DESCRIPTOR_TYPE_BOS

USB_B_DESCRIPTOR_TYPE_DEVICE_CAPABILITY

USB_B_DESCRIPTOR_TYPE_WIRELESS_ENDPOINT_COMP
USB_B_DESCRIPTOR_TYPE_WIRE_ADAPTER
USB_B_DESCRIPTOR_TYPE_RPIPE
USB_B_DESCRIPTOR_TYPE_CS_RADIO_CONTROL
USB_B_DESCRIPTOR_TYPE_PIPE_USAGE
Descriptor types from UAS specification.
USB_SETUP_PACKET_SIZE
Size of a USB control transfer setup packet in bytes.
USB_BM_REQUEST_TYPE_DIR_OUT
Bit masks belonging to the bmRequestType field of a setup packet.
USB_BM_REQUEST_TYPE_DIR_IN
USB_BM_REQUEST_TYPE_TYPE_STANDARD
USB_BM_REQUEST_TYPE_TYPE_CLASS
USB_BM_REQUEST_TYPE_TYPE_VENDOR
USB_BM_REQUEST_TYPE_TYPE_RESERVED
USB_BM_REQUEST_TYPE_TYPE_MASK
USB_BM_REQUEST_TYPE_RECIP_DEVICE
USB_BM_REQUEST_TYPE_RECIP_INTERFACE
USB_BM_REQUEST_TYPE_RECIP_ENDPOINT
USB_BM_REQUEST_TYPE_RECIP_OTHER
USB_BM_REQUEST_TYPE_RECIP_MASK
USB_B_REQUEST_GET_STATUS
Bit masks belonging to the bRequest field of a setup packet.
USB_B_REQUEST_CLEAR_FEATURE
USB_B_REQUEST_SET_FEATURE
USB_B_REQUEST_SET_ADDRESS
USB_B_REQUEST_GET_DESCRIPTOR

USB_B_REQUEST_SET_DESCRIPTOR

USB_B_REQUEST_GET_CONFIGURATION

USB_B_REQUEST_SET_CONFIGURATION

USB_B_REQUEST_GET_INTERFACE

USB_B_REQUEST_SET_INTERFACE

USB_B_REQUEST_SYNCH_FRAME

USB_W_VALUE_DT_DEVICE
    Bitmasks belonging to the wValue field of a setup packet.

USB_W_VALUE_DT_CONFIG

USB_W_VALUE_DT_STRING

USB_W_VALUE_DT_INTERFACE

USB_W_VALUE_DT_ENDPOINT

USB_W_VALUE_DT_DEVICE_QUALIFIER

USB_W_VALUE_DT_OTHER_SPEED_CONFIG

USB_W_VALUE_DT_INTERFACE_POWER

USB_SETUP_PACKET_INIT_SET_ADDR (setup_pkt_ptr, addr)
    Initializer for a SET_ADDRESS request.
    Sets the address of a connected device

USB_SETUP_PACKET_INIT_GET_DEVICE_DESC (setup_pkt_ptr)
    Initializer for a request to get a device’s device descriptor.

USB_SETUP_PACKET_INIT_GET_CONFIG (setup_pkt_ptr)
    Initializer for a request to get a device’s current configuration number.

USB_SETUP_PACKET_INIT_GET_CONFIG_DESC (setup_pkt_ptr, desc_index, desc_len)
    Initializer for a request to get one of the device’s current configuration descriptor.

    • desc_index indicates the configuration’s index number
    • Number of bytes of the configuration descriptor to get

USB_SETUP_PACKET_INIT_SET_CONFIG (setup_pkt_ptr, config_num)
    Initializer for a request to set a device’s current configuration number.
USB_SETUP_PACKET_INIT_SET_INTERFACE (setup_pkt_ptr, intf_num, alt_setting_num)
  Initializer for a request to set an interface’s alternate setting.

USB_SETUP_PACKET_INIT_GET_STR_DESC (setup_pkt_ptr, string_index, lang_id, desc_len)
  Initializer for a request to get a string descriptor.

USB_STANDARD_DESC_SIZE
  Size of dummy USB standard descriptor.

USB_DEVICE_DESC_SIZE
  Size of a USB device descriptor in bytes.

USB_CLASS_PER_INTERFACE
  Possible base class values of the bDeviceClass field of a USB device descriptor.

USB_CLASS_AUDIO

USB_CLASS_COMM

USB_CLASS_HID

USB_CLASS_PHYSICAL

USB_CLASS_STILL_IMAGE

USB_CLASS_PRINTER

USB_CLASS_MASS_STORAGE

USB_CLASS_HUB

USB_CLASS_CDC_DATA

USB_CLASS_CSCID

USB_CLASS_CONTENT_SEC

USB_CLASS_VIDEO

USB_CLASS_WIRELESS_CONTROLLER

USB_CLASS_PERSONAL_HEALTHCARE

USB_CLASS_AUDIO_VIDEO

USB_CLASS_BILLBOARD
Chapter 2. API Reference

USB_CLASS_USB_TYPE_C_BRIDGE

USB_CLASS_MISC

USB_CLASS_APP_SPEC

USB_CLASS_VENDOR_SPEC

USB_SUBCLASS_VENDOR_SPEC

Vendor specific subclass code.

USB_CONFIG_DESC_SIZE

Size of a short USB configuration descriptor in bytes.

Note: The size of a full USB configuration includes all the interface and endpoint descriptors of that configuration.

USB_BM_ATTRIBUTES_ONE

Bit masks belonging to the bmAttributes field of a configuration descriptor.

Must be set

USB_BM_ATTRIBUTES_SELFPOWER

Self powered

USB_BM_ATTRIBUTES_WAKEUP

Can wake-up

USB_BM_ATTRIBUTES_BATTERY

Battery powered

USB_IAD_DESC_SIZE

Size of a USB interface association descriptor in bytes.

USB_INTF_DESC_SIZE

Size of a USB interface descriptor in bytes.

USB_EP_DESC_SIZE

Size of a USB endpoint descriptor in bytes.

USB_B_ENDPOINT_ADDRESS_EP_NUM_MASK

Bit masks belonging to the bEndpointAddress field of an endpoint descriptor.

USB_B_ENDPOINT_ADDRESS_EP_DIR_MASK

USB_BM_ATTRIBUTES_XFERTYPE_MASK

Bit masks belonging to the bmAttributes field of an endpoint descriptor.
Chapter 2. API Reference

USB_BM_ATTRIBUTES_XFER_CONTROL
USB_BM_ATTRIBUTES_XFER_ISOC
USB_BM_ATTRIBUTES_XFER_BULK
USB_BM_ATTRIBUTES_XFER_INT
USB_BM_ATTRIBUTES_SYNCTYPE_MASK
USB_BM_ATTRIBUTES_SYNC_NONE
USB_BM_ATTRIBUTES_SYNC_ASYNC
USB_BM.Attributes_SYNC_ADAPTIVE
USB_BM_ATTRIBUTES_SYNC_SYNC
USB_BM_ATTRIBUTES_USAGETYPE_MASK
USB_BM_ATTRIBUTESUSAGE_DATA
USB_BM_ATTRIBUTESUSAGE_FEEDBACK
USB_BM_ATTRIBUTES_USAGE_IMPLICIT_FB

USB_EP_DESC_GET_XFERTYPE (desc_ptr)
    Macro helper to get information about an endpoint from its descriptor.

USB_EP_DESC_GET_EP_NUM (desc_ptr)
USB_EP_DESC_GET_EP_DIR (desc_ptr)
USB_EP_DESC_GET_MPS (desc_ptr)

USB_STR_DESC_SIZE
    Size of a short USB string descriptor in bytes.

Enumerations

enum usb_device_state_t
    USB2.0 device states.
    See Table 9-1 of USB2.0 specification for more details

Note: The USB_DEVICE_STATE_NOT_ATTACHED is not part of the USB2.0 specification, but is a catch
    all state for devices that need to be cleaned up after a sudden disconnection or port error.

Values:
enumerator **USB_DEVICE_STATE_NOT_ATTACHED**

The device was previously configured or suspended, but is no longer attached (either suddenly disconnected or a port error)

enumerator **USB_DEVICE_STATE_ATTACHED**

Device is attached to the USB, but is not powered.

enumerator **USB_DEVICE_STATEPowered**

Device is attached to the USB and powered, but has not been reset.

enumerator **USB_DEVICE_STATE_DEFAULT**

Device is attached to the USB and powered and has been reset, but has not been assigned a unique address.
Device responds at the default address.

enumerator **USB_DEVICE_STATE_ADDRESS**

Device is attached to the USB, powered, has been reset, and a unique device address has been assigned.
Device is not configured.

enumerator **USB_DEVICE_STATE_CONFIGURED**

Device is attached to the USB, powered, has been reset, has a unique address, is configured, and is not suspended. The host may now use the function provided by the device.

enumerator **USB_DEVICE_STATE_SUSPENDED**

Device is, at minimum, attached to the USB and is powered and has not seen bus activity for 3 ms. It may also have a unique address and be configured for use. However, because the device is suspended, the host may not use the device’s function.

Maintainers Notes

**Note:** For more details regarding the internal implementation details of the USB Host stack, please refer to [USB Host Maintainers Notes (Introduction)](USBHostMaintainersNotes(Introduction)).

**USB Host Maintainers Notes (Introduction)** This document contains information regarding the implementation details of the USB Host stack. This document is intended for the maintainers and third-party contributors of the USB Host stack. Users of the USB Host stack should refer to [USB Host](USBHost) instead.

**Warning:** The implementations details of the USB Host stack is categorized as private API. Thus, all layers (other than the USB Host Library) do not adhere to [ESP-IDF’s versioning scheme](ESP-IDFs%20versioning%20scheme) (i.e., breaking changes are permitted).

This document is split into the following sections:

**USB Host Maintainers Notes (Design Guidelines)**

**Design Considerations** The design of the Host Stack takes into account the following design considerations:

**Limited Hardware Resources:**
The embedded nature of Host Stack means limited hardware resources (such as memory and processing power) when compared to larger host systems.

**USB2.0 Chapter 10:**

Chapter 10 of the USB 2.0 specification specifies certain requirements of USB Host systems, in particular the required layers of the USB Host’s system software.

**Diverse Use Case Complexities:**

The embedded nature of the Host Stack also means a wide range of use cases with differing complexities. Some USB Host applications aim to only support a single vendor specific device, whereas other applications require support for a wide range of devices of different classes.

**Requirements**

Given the design considerations above, the Host Stack was designed with the following set of requirements:

**DMA Support**  Requirement: The Host Stack must support DMA.

The Host Stack must support DMA in order to reduce CPU’s workload. DMA support allows the automatic copying of USB transfer data to/from the Host Controller without CPU intervention. This is especially critical given the embedded nature of the CPU (i.e., lower CPU frequencies) and large maximum data payloads of USB transfers (e.g., 1023 bytes for isochronous transfers).

**Minimize Memory Copies**  Requirement: The Host Stack should minimize the amount of memory copies when passing data between layers.

Various data and objects (e.g., USB transfers) need to be passed between multiple layers of the Host Stack. The Host Stack should minimize the amount of memory copies that occur between layers by allocating the data’s/object’s memory once, and simply passing a pointer to that data/object between the layers. Therefore, the Host Stack requires some standardized data types shared across multiple layers (see USB2.0 Section 10.3.4).
Event Driven  Requirement: The Host Stack must allow for event driven operation (i.e., the Host Stack’s API must not be polling).

The Host Stack needs to support some CPU intensive use application scenarios such as video streaming (i.e., UVC class). Therefore, the Host Stack should minimize CPU usage by allowing for completely event driven operation, thus reserving the majority of CPU time for the application itself (i.e., video encoding/decoding in this case).

The Host Stack needs to communicate events across the layers using interrupts, callbacks, and FreeRTOS synchronization primitives (e.g., queues and semaphores).

No Task Creation  Requirement: All layers of the Host Stack below (and including) the Host Library layer must not create any tasks.

Task stacks are generally one of the most memory intensive parts of an ESP-IDF applications. Given the wide range of applications scenarios, the number of tasks created (and their stack sizes) can vary greatly. For example...

- applications that require low latency or high throughput (such as isochronous transfers) may choose to create a dedicated task to handle those transfers in order to minimize latency.
- applications that do not have strict latency requirements (such as bulk transfers) may choose to handle those transfers from a shared task in order to save some memory.

Therefore, all layers of the Host Stack below (and including) the Host Library layer must not create any tasks. Instead, these layers should expose handlers functions to be called from tasks created by the upper layers. Task creation will be delegated to the class driver or application layer.

Operable at Different Layers  Given the wide range of use case complexities, the Host Stack must be operable at different layers, allowing users to use the Host Stack at a lower layer (e.g., the HCD or HAL) or at a higher layer (e.g., a class driver).

Being operable at different layers allows the users to decide on the appropriate trade-offs between convenience, control, and optimization for their application when using the Host Stack. For example...

- Host Stack applications that support a dedicated custom device may want to use a lower level of abstraction for better optimization, control, and simpler API.
- Host Stack applications that need to support a wide range of device classes requires the full Host Stack so that device enumeration is automatically handled.

Coding Conventions  The Host Stack follows the following set of coding rules/guidelines for better code readability and maintainability:

Symbols Use Layer Name As Prefix  For each layer of the Host Stack, the symbols exposed by that layer (i.e., functions, types, macros) must be prefixed with that layer’s name. For example, the symbols exposed by the HCD layer will be prefixed `hcd_...`/`HCD_...`.

However, internal symbols (e.g., static functions) should not be prefixed with their layer’s name. This makes it easier to differentiate between internal and external symbols when modifying that layer’s source code.

Critical Section Functions Prefixed With _  In each layer of the Host Stack, there are various static functions that must be called inside a critical section. The names of these functions are prefixed with _ (e.g., `_func_called_from_crit()`) to make it easier for maintainers to differentiate which functions should be called from critical sections. For example...

```c
some_func(); // Called outside critical section
taskENTER_CRITICAL(&some_lock);
_some_func_crit(); // Called inside critical section. _ prefix makes it easier to differentiate
taskEXIT_CRITICAL(&some_lock);
```
Chapter 2. API Reference

**Grouping Structure Members by Locking Mechanism** Some layers of the Host Stack utilize multiple locking schemes (e.g., critical sections and task mutexes) to ensure thread safety, where each locking scheme offers a different level of protection. However, member variables of the same object can be protected by different locking scheme. Therefore, to clearly demarcate the different locking schemes and their associated variables, structure members are grouped by locking scheme as nested structures.

<table>
<thead>
<tr>
<th>Locking Scheme</th>
<th>Nested Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Sections</td>
<td>dynamic</td>
<td>Shared data accessed from both a task context and ISR context are protected by a critical section.</td>
</tr>
<tr>
<td>Task Mutexes</td>
<td>mux_protected</td>
<td>Shared data accessed from only a task context are protected by a FreeRTOS Mutex</td>
</tr>
<tr>
<td>Single Thread</td>
<td>single_thread</td>
<td>Data that is only ever accessed by the same task do not require the use of any locks.</td>
</tr>
<tr>
<td>Constant</td>
<td>constant</td>
<td>Constant data is set once during the object’s instantiation and never changed again. Thus, any task or ISR can freely the constant data without the use of locks, so long as the variable is never written to.</td>
</tr>
</tbody>
</table>

Grouping structure members by locking scheme makes the code more maintainable as it makes clear which locking scheme is required when accessing a particular member variable, as demonstrated in the code snippet below:

```c
typedef struct some_obj_t {
    // Accessing dynamic members requires critical section
    taskENTER_CRITICAL(&some_lock);
    obj.dynamic.varA = 1;
    taskEXIT_CRITICAL(&some_lock);

    // Accessing mutex protected members requires taking the mutex
    xSemaphoreTake(&some_mux, portMAX_DELAY);
    obj.mux_protected.varB = 1;
    xSemaphoreGive(&some_mux);

    // Accessing single thread members does not require locking so long as this is the only task to access it
    obj.single_thread.varC = 1;

    // Accessing constant members requires no locking. But only read access is allowed
    int local_var = obj.constant.varD;
}
```

**USB Host Maintainers Notes (Architecture)** The Host Stack is roughly split into multiple layers of abstraction, with each layer representing different USB concepts and a different level of USB Host operation. For example, a higher layer may present an abstraction of devices and application data transfers, whereas a lower layer may present an abstraction of endpoints and USB transfers.

**Layer Descriptions** The layers of the Host Stack are described in the following table. The layers are ordered from lowest layer (i.e., furthest away from the user) to highest layer (i.e., closest to the user).
### Table 7: Host Stack Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Controller (DWC_OTG)</td>
<td>N/A</td>
<td>This layer represents the USB Controller Hardware of the ESP32-S2. The API presented by this layer is the register interface of the controller.</td>
</tr>
<tr>
<td>LL</td>
<td>usbh_ll.h</td>
<td>The LL (Low Level) layer abstracts the basic register access of the USB controller according to ESP-IDF’s Hardware Abstraction API Guidelines. In other words, this layer provides APIs to access the controller’s registers and format/parse the controller’s DMA descriptors.</td>
</tr>
<tr>
<td>HAL</td>
<td>usbh_h.h, usbh_hal.h</td>
<td>The HAL (Hardware Abstraction Layer) abstracts the operating steps of the USB controller into functions according to ESP-IDF’s Hardware Abstraction API Guidelines. This layer also abstracts the controller’s host port and host channels, and provides APIs to operate the them.</td>
</tr>
<tr>
<td>HCD</td>
<td>hcd.h, hcd.c</td>
<td>The HCD (Host Controller Driver) acts as hardware agnostic API for all USB controllers (i.e., an API that can theoretically be used with any USB controller implementation). This layer also abstracts the root port (i.e., root hub) and USB pipes.</td>
</tr>
<tr>
<td>USBH and Hub Driver</td>
<td>usbh.h, usbh.c</td>
<td>The USBH (USB Host Driver) layer is equivalent to the USBD layer described in chapter 10 of the USB2.0 specification. The USBH presents an abstraction of USB devices, internally manages a list of connected devices (i.e., device pool), and also arbitrates device sharing between clients (i.e., tracks which endpoints are in use and also presents a shared endpoint 0).</td>
</tr>
<tr>
<td>Hub Driver</td>
<td>hub.h, hub.c</td>
<td>The Hub Driver layer acts as a special client of the USBH that is responsible for handling device attachment/detachment, and notifying the USBH of such events. For device attachment, the Hub Driver also handles the enumeration process as well.</td>
</tr>
<tr>
<td>USB Host Library</td>
<td>usb_host.h, usb_host.c</td>
<td>The USB Host Library layer is the lowest public API layer of the Host Stack and presents the concept of USB Host Clients. The abstraction of clients allows for multiple class drivers to coexist simultaneously (where each class roughly maps to a single client) and also acts as a mechanism for division of labor (where each client is responsible for its own processing and event handling).</td>
</tr>
<tr>
<td>Host Class Drivers</td>
<td>See the ESP-IDF Extra Components repository or the USB Host examples in ESP-IDF (via peripherals/usb/host).</td>
<td>The Host Class Drivers implement the host side of a particular device class (e.g., CDC, MSC, HID). The exposed API is specific to each class driver.</td>
</tr>
</tbody>
</table>

**Layer Dependencies** The Host Stack roughly follows a top to bottom hierarchy with inter-layer dependencies. Given layers A (highest), B, and C (lowest), the Host Stack has the following inter-layer dependency rules:

- a particular layer can use the API of any layer directly below (Layer A using layer B is allowed)
- a particular layer can use the API of any layer indirectly below (Layer A using layer C is allowed) i.e., skipping layers.
- a particular layer must not use the API of any layer above (Layer C using layer A/B is forbidden)
Note: Layer skipping is permitted in order to circumvent the need to repeat the same abstraction across multiple layers. For example, the abstraction of pipes are presented at the HCD layer but are used by multiple layers above.

USB Host Maintainers Notes (DWC_OTG Controller) The ESP32-S2 uses a DesignWare USB 2.0 On-the-Go Controller (henceforth referred to as DWC_OTG in this document) as its underlying hardware controller, where the DWC_OTG operates in Host Mode with Scatter/Gather DMA enabled.

Note: This section only summarizes the operation of the DWC_OTG operation in Host Mode at a high level. For full details of the DWC_OTG, refer to the DWC_OTG Databook and Programming Guide.

Host Mode Operating Model A simplified version of the operating model of the DWC_OTG in Host Mode is illustrated in the diagram below. The diagram contains some of the key concepts and terms regarding DWC_OTG Host Mode.

Note: Refer to Databook section 2.1.4 (Host Architecture) for more details

Host Port The Host Port represents the single USB port provided by the DWC_OTG (in USB terms, this can be thought a single USB port of the root hub of the bus). The Host Port can only connect to a single device, though more devices can be connected via hub devices.

The Host Port is responsible for:

- detecting direct device connections/disconnections
- detecting the speed of the directly connected device
- issuing various bus signals (such as suspend, resume, reset)

Host Channels In Host Mode, the DWC_OTG uses channels to communicate with device endpoints, where one channel maps to a particular endpoint (in USB terms, channels are the hardware representation of pipes). For example, a channel will map to EP 1 OUT. Each channel has its own set of CSRs (Control and Status Registers) so that they can independently configured and controlled independently. A channel’s CSRs are used to:

- specify the details of the channel’s target endpoint (e.g., device address, EP number, transfer type, direction)
- start a transfer on the channel (e.g., by setting up DMA descriptors)

When using Scatter/Gather DMA, transfers on Host Channels are completely event driven. Users simply fill out the appropriate DMA descriptors, fill in the channel’s CSRs, then enable the channel. The channel will then generate an interrupt when the transfer completes.
Data FIFOs  In Host Mode, the DWC_OTG uses multiple FIFOs as a staging area for the data payloads of USB transfers. When using DMA, the DMA engine will copy data between the TX/RX FIFOs and ESP32-S2’s internal memory:

- For an OUT transfer, the transfer’s data payload is copied from main memory to one of the TX FIFOs by DMA. The MAC Layer will then transmit that data payload in accordance to USB packet formatting.
- For an IN transfer, the MAC Layer will parse the received USB packet and store the received data payload in the RX FIFO. The data is then copied to main memory by DMA.

The destination FIFO depends on the direction and transfer type of the channel:

- All IN channel data goes to the RX FIFO
- All non-periodic (i.e., Control and Bulk) OUT channel data goes to the Non-periodic TX (NPTX) FIFO
- All periodic (i.e., Interrupt and Isochronous) OUT channel data goes to the Periodic TX (PTX) FIFO

Note: The separation of non-periodic and periodic OUT channels to the NPTX and PTX FIFOs is due to the periodic nature of Interrupt and Isochronous endpoints (specified by the bInterval value of the endpoint). The DWC_OTG automatically schedules these periodic transfers, thus a separate PTX FIFO allows these periodic transfers to be staged separately.

DMA Engine  The DMA engine is responsible for copying data between the FIFOs and main memory. In Host Mode Scatter/Gather DMA, a particular channel can carry out multiple transfers automatically without software intervention. The following diagram illustrates the DWC_OTG Host Mode Scatter/Gather DMA Memory Structures.

- Each USB transfer is described by a Queue Transfer Descriptor (QTD). Each QTD consists of:
  - A 32-bit Buffer Status Quadlet specifying details of the transfer, and also reports the status of the transfer on completion. The Buffer Status Quadlet has bits to specify whether the QTD should generate an interrupt and/or halt the channel on completion.
  - A 32-bit pointer to the data buffer containing the data payload for OUT transfers, or an empty buffer used to store the data payload for IN transfers.
- The data payload of each QTD can be larger than the MPS (Maximum Packet Size) of its target endpoint. The DWC_OTG hardware automatically handles splitting of the transfer into multiple transactions.
- Multiple QTDs can be placed into a single QTD List. A channel will then execute each QTD in the list automatically, and optionally loop back around if configured to do so.
- Before a channel starts data transfer, it is configured with a QTD list (and QTD list length). Once the channel is enabled, USB transfers are executed automatically by the hardware.
- A channel can generate interrupts (configurable) on completion of a particular QTD, or an entire QTD list.
Chapter 2. API Reference

Note: Refer to Programming Guide section 6.2.1 (Descriptor Memory Structures) for more details

**Hardware Configuration** The DWC_OTG IP is configurable. The notable Host related configurations of the ESP32-S2’s DWC_OTG are listed below:

<table>
<thead>
<tr>
<th>Description</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host and Device Mode support with OTG</td>
<td>OTG_MODE = 0</td>
</tr>
<tr>
<td>Full Speed (FS) and Low Speed (LS) support</td>
<td>OTG_FSPHY_INTERFACE = 1,</td>
</tr>
<tr>
<td></td>
<td>OTG_HSPHY_INTERFACE = 0</td>
</tr>
<tr>
<td>Internal DMA controller with Scatter/Gather DMA</td>
<td>OTG_ARCHITECTURE = 2,</td>
</tr>
<tr>
<td></td>
<td>OTG_EN_DESC_DMA = 1</td>
</tr>
<tr>
<td>FS Hubs are supported but HS Hub are not (i.e., split transfers not supported)</td>
<td>OTG_SINGLE_POINT = 0</td>
</tr>
<tr>
<td>8 Host Mode channels</td>
<td>OTG_NUM_HOST_CHAN = 8</td>
</tr>
<tr>
<td>All transfer types supported, including ISOC and INTR OUT transfers</td>
<td>OTG_EN_PERIO_HOST = 1</td>
</tr>
<tr>
<td>Dynamically sized Data FIFO of 1024 bytes (256 lines)</td>
<td>OTG_DFIFO_DYNAMIC = 1,</td>
</tr>
<tr>
<td></td>
<td>OTG_DFIFO_DEPTH = 256</td>
</tr>
</tbody>
</table>

**Scatter/Gather DMA Transfer** The basic operating procedure for Host channels transfers consists of the following steps:

1. Prepare data buffers, QTDs, and QTD list. In particular, which QTDs should halt the channel (and generate an interrupt) on completion.
2. Set channel/endpoint characteristics via CSRs (such as EP address, transfer type, EP MPS etc).
3. Set channel’s QTD list related CSRs (such as QTD list pointer and QTD list length) and channel interrupt CSRs.
4. Enable the channel. Transfers are now handled automatically by hardware using DMA.
5. The Channel generates an interrupt on a channel event (e.g., QTD completion or channel error).
6. Parse the channel interrupt to determine what event occurred.
7. Parse the QTDs to determine the result of each individual transfer.

However, there are some minor differences in channel operation and QTD list usage depending on the transfer type.

**Bulk** Bulk transfers are a simplest. Each QTD represents a bulk transfer of a particular direction, where the DWC_OTG automatically splits a particular QTD into multiple MPS sized transactions. Thus it is possible to fill a QTD list with multiple bulk transfers, and have the entire list executed automatically (i.e., only interrupt on completion of the last QTD).

**Control** Control transfers are more complicated as they are bi-directional (i.e., each control transfer stage can have a different direction). Thus, a separate QTD is required for each stage, and each QTD must halt the channel on completion. Halting the channel after each QTD allows changing the channel’s direction to be changed by reconfiguring the channel’s CSRs. Thus a typical control transfer consists of 3 QTDs (one for each stage).

**Interrupt** In accordance with the USB2.0 specification, interrupt transfers executes transactions at the endpoints specified service period (i.e., bInterval). A particular interrupt endpoint may not execute more than one interrupt transaction within a service period. The service period is specified in number of microframes/frames, thus a particular interrupt endpoint will generally execute one transaction every Nth microframe/frame until the transfer is complete. For interrupt channels, the service period of a particular channel (i.e., bInterval) is specified via the Host Frame List (see section 6.5 of programming guide for more details).
**Note:** HS USB allows an interrupt endpoint to have 3 interrupt transactions in a single microframe. See USB2.0 specification section 5.7.3 (Interrupt Transfer Packet Size Constraints) for more details.

Thus, interrupt transfers in Host Mode Scatter/Gather DMA have the following peculiarities:

- If a QTD payload is larger than the endpoint’s MPS, the channel will automatically split the transfer into multiple MPS sized transactions (similar to bulk transfers). However, each transaction is executed at endpoint’s specified service period (i.e., one transaction per bInterval) until the transfer completes.
- For Interrupt IN transfers, if a short packet is received (i.e., transaction’s data payload is < MPS), this indicates that the endpoint has no more data to send. In this case:
  - the channel generates an extra channel interrupt even if the transfer’s QTD did not set the IOC (interrupt on complete) bit.
  - however, the channel is not halted even if this extra channel interrupt is generated.
  - software must then use this extra interrupt to manually halt the interrupt channel (thus canceling any remaining QTDs in the QTD list).

**Note:** Due to the interrupt transfer peculiarities, it may be easier for software allocate a QTD for each transaction instead of an entire transfer.

---

**Isochronous** In accordance with the USB2.0 specification, isochronous transfers executes transactions at the endpoints specified service period (i.e., bInterval) in order to achieve a constant rate of data transfer. A particular isochronous endpoint may not execute more than one isochronous transaction within a service period. The service period is specified in number of microframes/frames, thus a particular isochronous endpoint will generally execute one transaction every Nth microframe/frame until the transfer is complete. For isochronous channels, the service period of a particular channel (i.e., bInterval) is specified via the Host Frame List (see section 6.5 of programming guide for more details).

However, unlike interrupt transactions, isochronous transactions are not retried on failure (or NAK), due to the need to maintain the constant data rate.

**Note:** HS USB allows an isochronous endpoint to have 3 interrupt transactions in a single microframe. See USB2.0 specification section 5.6.3 (Isochronous Transfer Packet Size Constraints) for more details.

Thus, isochronous transfers in Host Mode Scatter/Gather DMA have the following peculiarities:

- A QTD must be allocated for each microframe/frame. However, non-service service period QTDs should be left blank (i.e., only every Nth QTD should be filled if the channel’s service period is every Nth microframe/frame).
- Each filled QTD must represent a single transaction instead of a transfer.
- Because isochronous transactions are not retried on failure, the status each completed QTD must be checked.

**Supplemental Notes** Some of the DWC_OTG’s behaviors are not mentioned in the Databook or Programming Guide. This section describes some of those behaviors that are relevant to the Host stack’s implementation.

**Port Errors Do Not Trigger a Channel Interrupt** If a port error occurs (such as a sudden disconnection or port over-current) while there are one or more active channels...

- The active channels remains active (i.e., HCCHAR.ChEna remains set) and no channel interrupts are generated.
- Channels could in theory be disabled by setting HCCHAR.ChDis, but this does not work for Isochronous channels as the channel disabled interrupt is never generated.

Therefore, on port errors, a controller soft reset should be used to ensure all channels are disabled.
Chapter 2. API Reference

Port Reset Interrupts

• When the DWC_OTG issues a reset signal on its port, and during the reset signal the device disconnects, the disconnection interrupt (i.e., HPRT.PrtConnDet) is not generated until the reset is deasserted.
• When resetting an already enabled port (i.e., HPRT.PrtEna) such as a second reset during enumeration or a run-time reset, a Port Enable/Disable Change interrupt (i.e., HPRT.PrtEnChng) is generated both on the assertion and deassertion of the reset signal.

Todo:
• USB Host Maintainers Notes (HAL & LL)
• USB Host Maintainers Notes (HCD)
• USB Host Maintainers Notes (USBH)
• USB Host Maintainers Notes (Hub)
• USB Host Maintainers Notes (USB Host Library)

Introduction

The ESP-IDF USB Host Stack allows the ESP32-S2 to operate as a USB Host. Operating as a USB Host allows the ESP32-S2 to communicate with a wide range of USB devices. However, most USB Host Stack implementations do not run on embedded hardware (i.e., runs on PCs and smartphones), thus have comparatively more resources (i.e., memory and CPU speed).

The implementation of the ESP-IDF USB Host Stack (henceforth referred to as the Host Stack) takes into account the embedded nature of the ESP32-S2 which is reflected in various aspects of the Host Stack’s design.

Features & Limitations

The Host Stack currently supports the following notable features:

• Support FS (Full Speed) and LS (Low Speed) devices
• Support all transfer types (Control, Bulk, Isochronous, and Interrupt)
• Automatically enumerates connected devices
• Allows multiple class drivers (i.e., Clients of the USB Host Library) to run simultaneously and share the same device (i.e., composite devices).

The Host Stack currently has the following notable limitations:

• No HS (High Speed) support
• No Hub support (currently only supports a single device)

Code examples for this API section are provided in the peripherals directory of ESP-IDF examples.

2.6 Project Configuration

2.6.1 Introduction

The esp-idf-kconfig package that ESP-IDF uses is based on kconfiglib, which is a Python extension to the Kconfig system. Kconfig provides a compile-time project configuration mechanism and offers configuration options of several types (e.g., integers, strings, and booleans). Kconfig files specify dependencies between options, default values of options, the way options are grouped together, etc.

For the full list of available features, please see Kconfig and kconfiglib extensions.

2.6.2 Project Configuration Menu

Application developers can open a terminal-based project configuration menu with the idf.py menuconfig build target.
Chapter 2. API Reference

After being updated, this configuration is saved in the `sdkconfig` file under the project root directory. Based on `sdkconfig`, application build targets will generate the `sdkconfig.h` file under the build directory, and will make the `sdkconfig` options available to the project build system and source files.

2.6.3 Using `sdkconfig.defaults`

In some cases, for example, when the `sdkconfig` file is under revision control, it may be inconvenient for the build system to change the `sdkconfig` file. The build system offers a solution to prevent it from happening, which is to create the `sdkconfig.defaults` file. This file is never touched by the build system, and can be created manually or automatically. It contains all the options which matter to the given application and are different from the default ones. The format is the same as that of the `sdkconfig` file. `sdkconfig.defaults` can be created manually when one remembers all the changed configuration, or it can be generated automatically by running the `idf.py save-defconfig` command.

Once `sdkconfig.defaults` is created, `sdkconfig` can be deleted or added to the ignore list of the revision control system (e.g., the `.gitignore` file for `git`). Project build targets will automatically create the `sdkconfig.defaults` file, populate it with the settings from the `sdkconfig.defaults` file, and configure the rest of the settings to their default values. Note that during the build process, settings from `sdkconfig.defaults` will not override those already in `sdkconfig`. For more information, see Custom `Sdkconfig Defaults`.

2.6.4 Kconfig Format Rules

Format rules for Kconfig files are as follows:

- Option names in any menus should have consistent prefixes. The prefix currently should have at least 3 characters.
- The unit of indentation should be 4 spaces. All sub-items belonging to a parent item are indented by one level deeper. For example, `menu` is indented by 0 spaces, `config menu` by 4 spaces, `help in config` by 8 spaces, and the text under `help` by 12 spaces.
- No trailing spaces are allowed at the end of the lines.
- The maximum length of options is 40 characters.
- The maximum length of lines is 120 characters.

Note: The `help` section of each config in the menu is treated as reStructuredText to generate the reference documentation for each option.

Format Checker

`kconfcheck` tool in `esp-idf-kconfig` package is provided for checking Kconfig files against the above format rules. The checker checks all Kconfig and Kconfig.projbuild files given as arguments, and generates a new file with suffix `.new` with some suggestions about how to fix issues (if there are any). Please note that the checker cannot correct all format issues and the responsibility of the developer is to final check and make corrections in order to pass the tests. For example, indentations will be corrected if there is not any misleading formatting, but it cannot come up with a common prefix for options inside a menu.

The `esp-idf-kconfig` package is available in ESP-IDF environments, where the checker tool can be invoked by running command `python -m kconfcheck <path_to_kconfig_file>`.

For more information, please refer to `esp-idf-kconfig` package documentation.

2.6.5 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:
Chapter 2. API Reference

1. kconfgen is used by the tool chain to pre-process sdkconfig files before anything else. For example, menuconfig would read them, so the settings for old options is kept and not ignored.

2. kconfgen 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. Renames that should only appear for a single target can be placed in a target-specific rename file sdkconfig.rename.TARGET, where TARGET is the target name, e.g., sdkconfig.rename.esp32s2.

3. kconfgen post-processes sdkconfig files and generates all build outputs (sdkconfig.h, sdkconfig.cmake, and auto.conf) by adding a list of compatibility statements, i.e., the values of old options are set for new options after modification. If users still use old options in their code, this will prevent it from breaking.

4. Deprecated options and their replacements are automatically generated by kconfgen.

2.6.6 Configuration Options Reference

Subsequent sections contain the list of available ESP-IDF options automatically generated from Kconfig files. Note that due to dependencies between options, some options listed here may not be visible by default in menuconfig.

By convention, all option names are upper-case letters 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 the sdkconfig and sdkconfig.h files will have CONFIG_ENABLE_FOO defined. In the following sections, option names are also prefixed with CONFIG_, same as in the source code.

Build type

Contains:

- CONFIG_APP_BUILD_TYPE
- CONFIG_APP_BUILD_TYPE_PURE_RAM_APP
- CONFIG_APP_REPRODUCIBLE_BUILD
- CONFIG_APP_NO_BLOBS

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 ESP-IDF 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 or UART. Note that since IRAM and DRAM sizes are very limited, it is not possible to build any complex application this way. However for some kinds of testing and debugging, this option may provide faster iterations, since the application does not need to be written into flash.

Note: when APP_BUILD_TYPE_RAM is selected and loaded with JTAG, 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 (for ESP32):

```gdb
# 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-gdbbuild/app-name.elf-x gdbinit

Example gdbinit files for other targets can be found in tools/test_apps/system/gdb_loadable_elf/

When loading the BIN with UART, the ROM will jump to ram and run the app after finishing the ROM startup code, so there’s no additional startup initialization required. You can use the load_ram in esptool.py to load the generated .bin file into ram and execute.

Example: esptool.py -chip {chip} -p {port} -b {baud} -no-stub load_ram {app.bin}

Recommended sdkconfig.defaults for building loadable ELF files is as follows. CONFIG_APP_BUILD_TYPE_RAM is required, other options help reduce application memory footprint.

CONFIG_APP_BUILD_TYPE_RAM=y CONFIG_VFS_SUPPORT_TERMIOS= CONFIG_NEWLIB_NANO_FORMAT=y CONFIG_ESP_SYSTEM_PANIC_PRINT_HALT=y CONFIG_ESP_DEBUG_STUBS_ENABLE=CONFIG_ESP_ERR_TO_NAME_LOOKUP=

Available options:

• Default (binary application + 2nd stage bootloader) (CONFIG_APP_BUILD_TYPE_APP_2NDBOOT)
• Build app runs entirely in RAM (EXPERIMENTAL) (CONFIG_APP_BUILD_TYPE_RAM)

CONFIG_APP_BUILD_TYPEPURE_RAM_APP

Build app without SPI_FLASH/PSRAM support (saves ram)

Found in: Build type

If this option is enabled, external memory and related peripherals, such as Cache, MMU, Flash and PSRAM, won’t be initialized. Corresponding drivers won’t be introduced either. Components that depend on the spi_flash component will also be unavailable, such as app_update, etc. When this option is enabled, about 26KB of RAM space can be saved.

CONFIG_APP_REPODUCIBLE_BUILD

Enable reproducible build

Found in: Build type

If enabled, all date, time, and path information would be eliminated. A .gdbinit file would be create automatically. (or will be append if you have one already)

Default value:
• No (disabled)

CONFIG_APP_NO_BLOBS

No Binary Blobs

Found in: Build type

If enabled, this disables the linking of binary libraries in the application build. Note that after enabling this Wi-Fi/Bluetooth will not work.

Default value:
• No (disabled)
Bootloader config

Contains:

- `CONFIG_BOOTLOADER_LOG_LEVEL`
- Bootloader manager
- `CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION`
- `CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE`
- `CONFIG_BOOTLOADER_REGION_PROTECTION_ENABLE`
- `CONFIG_BOOTLOADER_APP_TEST`
- `CONFIG_BOOTLOADER_FACTORY_RESET`
- `CONFIG_BOOTLOADER_HOLD_TIME_GPIO`
- `CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC`
- Serial Flash Configurations
- `CONFIG_BOOTLOADER_SKIP_VALIDATE_ALWAYS`
- `CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON`
- `CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP`
- `CONFIG_BOOTLOADER_WDT_ENABLE`
- `CONFIG_BOOTLOADER_VDDSDIO_BOOST`

Bootloader manager

Contains:

- `CONFIG_BOOTLOADER_PROJECT_VER`
- `CONFIG_BOOTLOADER_COMPILE_TIME_DATE`

`CONFIG_BOOTLOADER_COMPILE_TIME_DATE`

Use time/date stamp for bootloader

*Found in: Bootloader config > Bootloader manager*

If set, then the bootloader will be built with the current time/date stamp. It is stored in the bootloader description structure. If not set, time/date stamp will be excluded from bootloader image. This can be useful for getting the same binary image files made from the same source, but at different times.

`CONFIG_BOOTLOADER_PROJECT_VER`

Project version

*Found in: Bootloader config > Bootloader manager*

Project version. It is placed in “version” field of the esp_bootloader_desc structure. The type of this field is “uint32_t”.

*Range:*
- from 0 to 4294967295

*Default value:*
- 1

`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) (CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION_SIZE)
- Debug (-Og) (CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION_DEBUG)
- Optimize for performance (-O2) (CONFIG_BOOTLOADER_COMPILER_OPTIMIZATION_PERF)
- Debug without optimization (-O0) (CONFIG_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 (CONFIG_BOOTLOADER_LOG_LEVEL_NONE)
- Error (CONFIG_BOOTLOADER_LOG_LEVEL_ERROR)
- Warning (CONFIG_BOOTLOADER_LOG_LEVEL_WARN)
- Info (CONFIG_BOOTLOADER_LOG_LEVEL_INFO)
- Debug (CONFIG_BOOTLOADER_LOG_LEVEL_DEBUG)
- Verbose (CONFIG_BOOTLOADER_LOG_LEVEL_VERBOSE)

Serial Flash Configurations

Contains:

- CONFIG_BOOTLOADER_FLASH_DC_AWARE
- CONFIG_BOOTLOADER_FLASH_XMC_SUPPORT

CONFIG_BOOTLOADER_FLASH_DC_AWARE

Allow app adjust Dummy Cycle bits in SPI Flash for higher frequency (READ HELP FIRST)

*Found in: Bootloader config > Serial Flash Configurations*

This will force 2nd bootloader to be loaded by DOUT mode, and will restore Dummy Cycle setting by resetting the Flash

CONFIG_BOOTLOADER_FLASH_XMC_SUPPORT

Enable the support for flash chips of XMC (READ DOCS FIRST)

*Found in: Bootloader config > Serial Flash Configurations*

Perform the startup flow recommended by XMC. Please consult XMC for the details of this flow. XMC chips will be forbidden to be used, when this option is disabled.

DON’T DISABLE THIS UNLESS YOU KNOW WHAT YOU ARE DOING.

 Default value:

- Yes (enabled)
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 (CONFIG_BOOTLOADER_VDDSDIO_BOOST_1_8V)
- 1.9V (CONFIG_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 held at the configured level while device starts up. See settings below.

Default value:
- No (disabled)

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 (note that on some SoCs, not all pins have an internal pull-up, consult the hardware datasheet for details.) To trigger a factory reset, this GPIO must be held high or low (as configured) on startup.

Range:
- from 0 to 44 if CONFIG_BOOTLOADER_FACTORY_RESET

Default value:
- 4 if CONFIG_BOOTLOADER_FACTORY_RESET

CONFIG_BOOTLOADER_FACTORY_RESET_PIN_LEVEL

Factory reset GPIO level

Found in: Bootloader config > CONFIG_BOOTLOADER_FACTORY_RESET

Pin level for factory reset, can be triggered on low or high.

Available options:

- Reset on GPIO low (CONFIG_BOOTLOADER_FACTORY_RESET_PIN_LOW)
- Reset on GPIO high (CONFIG_BOOTLOADER_FACTORY_RESET_PIN_HIGH)
**CONFIG_BOOTLOADER_OOTA_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.

**Default value:**
- “nvs” if CONFIG_BOOTLOADER_FACTORY_RESET

**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.

**Range:**
- from 0 to 39 if CONFIG_BOOTLOADER_APP_TEST

**Default value:**
- 18 if CONFIG_BOOTLOADER_APP_TEST

**CONFIG_BOOTLOADER_APP_TEST_PIN_LEVEL**

App test GPIO level

*Found in: Bootloader config > CONFIG_BOOTLOADER_APP_TEST*

Pin level for app test, can be triggered on low or high.

Available options:

- Enter test app on GPIO low (CONFIG_BOOTLOADER_APP_TEST_PIN_LOW)
- Enter test app on GPIO high (CONFIG_BOOTLOADER_APP_TEST_PIN_HIGH)
**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.

*Default value:*
- 5 if `CONFIG_BOOTLOADER_FACTORY_RESET || CONFIG_BOOTLOADER_APP_TEST`

**CONFIG_BOOTLOADER_REGION_PROTECTION_ENABLE**

Enable protection for unmapped memory regions

*Found in: Bootloader config*

Protects the unmapped memory regions of the entire address space from unintended accesses. This will ensure that an exception will be triggered whenever the CPU performs a memory operation on unmapped regions of the address space.

*Default value:*
- Yes (enabled)

**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 RTC_CLK_SRC (INTERNAL_RC or EXTERNALCRYSTAL).

*Default value:*
- Yes (enabled)

**CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE**

Allows RTC watchdog disable in user code

*Found in: Bootloader config > CONFIG_BOOTLOADER_WDT_ENABLE*

If this option is set, the ESP-IDF app must explicitly reset, feed, or disable the rtc_wdt in the app’s own code. If this option is not set (default), then rtc_wdt will be disabled by ESP-IDF before calling the app_main() function.

Use function rtc_wdt_feed() for resetting counter of rtc_wdt. Use function rtc_wdt_disable() for disabling rtc_wdt.

*Default value:*
- No (disabled)

**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.

**Range:**
- from 0 to 120000

**Default value:**
- 9000

**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_OOTA_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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled) if **CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE**

**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).

**Default value:**
- 0 if **CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK**
**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 ESP32-S2. This determines how many times the security version can be increased.

**Range:**
- from 1 to 16 if `CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK`

**Default value:**
- 16 if `CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK`

**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 allows to emulate read/write operations with all eFuses and efuse secure version. It allows to test anti-rollback implementation without permanent write eFuse bits. There should be an entry in partition table with following details: `emul_efuse, data, efuse, , 0x2000`.

This option enables: EFUSE_VIRTUAL and EFUSE_VIRTUAL_KEEP_IN_FLASH.

**Default value:**
- No (disabled) if `CONFIG_BOOTLOADER_APP_ANTI_ROLLBACK`

**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.

It is possible to enable this option with Secure Boot if “allow insecure options” is enabled, however it’s strongly recommended to NOT enable it as it may allow a Secure Boot bypass.

**Default value:**
- No (disabled) if `CONFIG_SECURE_BOOT` && `CONFIG_SECURE_BOOT_INSECURE`

**CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON**

Skip image validation from power on reset (READ HELP FIRST)

*Found in: Bootloader config*

Some applications need to boot very quickly from power on. By default, the entire app binary is read from flash and verified which takes up a significant portion of the boot time.

Enabling this option will skip validation of the app when the SoC boots from power on. Note that in this case it’s not possible for the bootloader to detect if an app image is corrupted in the flash, therefore it’s not possible to safely fall back to a different app partition. Flash corruption of this kind is unlikely but can happen if there is a serious firmware bug or physical damage.
Following other reset types, the bootloader will still validate the app image. This increases the chances that flash corruption resulting in a crash can be detected following soft reset, and the bootloader will fall back to a valid app image. To increase the chances of successfully recovering from a flash corruption event, keep the option `BOOTLOADER_WDT_ENABLE` enabled and consider also enabling `BOOTLOADER_WDT_DISABLE_IN_USER_CODE` - then manually disable the RTC Watchdog once the app is running. In addition, enable both the Task and Interrupt watchdog timers with reset options set.

**Default value:**
- No (disabled)

**CONFIG_BOOTLOADER_SKIP_VALIDATE_ALWAYS**

Skip image validation always (READ HELP FIRST)

*Found in: Bootloader config*

Selecting this option prevents the bootloader from ever validating the app image before booting it. Any flash corruption of the selected app partition will make the entire SoC unbootable.

Although flash corruption is a very rare case, it is not recommended to select this option. Consider selecting “Skip image validation from power on reset” instead. However, if boot time is the only important factor then it can be enabled.

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- 0 if `CONFIG_BOOTLOADER_CUSTOM_RESERVE_RTC`

**Security features**

Contains:

- `CONFIG_SECURE_BOOT_INSECURE`
- `CONFIG_SECURE_SIGNED_APPS_SCHEME`
- `CONFIG_SECURE_SIGNED_ON_BOOT_NO_SECURE_BOOT`
- `CONFIG_SECURE_FLASH_CHECK_ENC_EN_IN_APP`
- `CONFIG_SECURE_BOOT_ECDSA_KEY_LEN_SIZE`
• **CONFIG_SECURE_BOOT_ENABLE_AGGRESSIVE_KEY_REVOKE**
• **CONFIG_SECURE_FLASH_ENC_ENABLED**
• **CONFIG_SECURE_BOOT**
• **CONFIG_SECURE_BOOT_FLASH_BOOTLOADER_DEFAULT**
• **CONFIG_SECURE_BOOTLOADER_KEY_ENCODING**
  • Potentially insecure options
• **CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT**
• **CONFIG_SECURE_BOOT_VERIFICATION_KEY**
• **CONFIG_SECURE_BOOTLOADER_MODE**
• **CONFIG_SECURE_BOOTBUILD_SIGNED_BINARIES**
• **CONFIG_SECURE_UART_ROM_DL_MODE**
• **CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT**

**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 secure boot versions:

1. **Secure boot V1**
   • Legacy custom secure boot scheme. Supported in ESP32 SoC.
2. **Secure boot V2**
   • RSA based secure boot scheme. Supported in ESP32-ECO3 (ESP32 Chip Revision 3 onwards), ESP32-S2, ESP32-C3, ESP32-S3 SoCs.
   • ECDSA based secure boot scheme. Supported in ESP32-C2 SoC.

Available options:

• **ECDSA (CONFIG_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 (CONFIG_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.
• **ECDSA (V2) (CONFIG_SECURE_SIGNED_APPS_ECDSA_V2_SCHEME)**
  For Secure boot V2 (e.g., ESP32-C2 SoC), appends ECDSA based signature block to the application. Refer to documentation before enabling.

**CONFIG_SECURE_BOOT_ECDSA_KEY_LEN_SIZE**

ECDSA key size

*Found in: Security features*

Select the ECDSA key size. Two key sizes are supported.
• 192 bit key using NISTP192 curve
• 256 bit key using NISTP256 curve (Recommended)

The advantage of using 256 bit key is the extra randomness which makes it difficult to be bruteforced compared to 192 bit key. At present, both key sizes are practically implausible to bruteforce.

Available options:

• Using ECC curve NISTP192 (CONFIG_SECURE_BOOT_ECDSA_KEY_LEN_192_BITS)
• Using ECC curve NISTP256 (Recommended) (CONFIG_SECURE_BOOT_ECDSA_KEY_LEN_256_BITS)

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.

Default value:
• No (disabled) if CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT & & CONFIG_SECURE_SIGNED_APPS_ECDSA_SCHEME

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.

Default value:
• Yes (enabled) if CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT

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.

Default value:
• No (disabled)
Chapter 2. API Reference

**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 / ECDSA based secure boot scheme.

- RSA based scheme is supported in ESP32 (Revision 3 onwards), ESP32-S2, ESP32-C3 (ECO3), ESP32-S3.
- ECDSA based scheme is supported in ESP32-C2 SoC.

Please note that, RSA or ECDSA secure boot is property of specific SoC based on its HW design, supported crypto accelerators, die-size, cost and similar parameters. Please note that RSA scheme has requirement for bigger key sizes but at the same time it is comparatively faster than ECDSA verification.

Secure Boot V1 is the AES based (custom) secure boot scheme supported in ESP32 SoC.

Available options:

- Enable Secure Boot version 1 (CONFIG_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 (CONFIG_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 (CONFIG_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 (CONFIG_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.

**Default value:**

- “secure_boot_signing_key.pem” if CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES

**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.

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_BOOT_ENABLE_AGGRESSIVE_KEY_REVOKE**

Enable Aggressive key revoke strategy

*Found in:* Security features

If this option is set, ROM bootloader will revoke the public key digest burned in efuse block if it fails to verify the signature of software bootloader with it. Revocation of keys does not happen when enabling secure boot. Once secure boot is enabled, key revocation checks will be done on subsequent boot-up, while verifying the software bootloader

This feature provides a strong resistance against physical attacks on the device.

**NOTE:** Once a digest slot is revoked, it can never be used again to verify an image. This can lead to permanent bricking of the device, in case all keys are revoked because of signature verification failure.

**Default value:**

- No (disabled) if CONFIG_SECURE_BOOT

**CONFIG_SECURE_BOOT_FLASH_BOOTLOADER_DEFAULT**

Flash bootloader along with other artifacts when using the default flash command

*Found in:* Security features

When Secure Boot V2 is enabled, by default the bootloader is not flashed along with other artifacts like the application and the partition table images, i.e. bootloader has to be seperately flashed using the
command `idf.py bootloader flash`, whereas, the application and partition table can be flashed using the command `idf.py flash` itself. Enabling this option allows flashing the bootloader along with the other artifacts by invocation of the command `idf.py flash`.

If this option is enabled make sure that even the bootloader is signed using the correct secure boot key, otherwise the bootloader signature verification would fail, as hash of the public key which is present in the bootloader signature would not match with the digest stored into the efuses and thus the device will not be able to boot up.

**Default value:**
- No (disabled) if `CONFIG_SECURE_BOOT_V2_ENABLED` && `CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES`

**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 espfuse.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) (CONFIG_SECURE_BOOTLOADER_KEY_ENCODING_256BIT)
- 3/4 encoding (192 bit key) (CONFIG_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.

**Default value:**
- No (disabled) if `CONFIG_SECURE_BOOT`

**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.

**Default value:**
- No (disabled)
CONFIG_SECURE_FLASH_ENCRYPTION_KEYSIZE

Size of generated XTS-AES key

*Found in: Security features > CONFIG_SECURE_FLASH_ENC_ENABLED*

Size of generated XTS-AES key.

- AES-128 uses a 256-bit key (32 bytes) derived from 128 bits (16 bytes) burned in half Efuse key block. Internally, it calculates SHA256(128 bits)
- 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 key derived from 128 bits (SHA256(128 bits)) (CONFIG_SECURE_FLASH_ENCRYPTION_AES128_DERIVED)
- AES-128 (256-bit key) (CONFIG_SECURE_FLASH_ENCRYPTION_AES128)
- AES-256 (512-bit key) (CONFIG_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 ROM download mode to perform flash encryption operations (plaintext is sent to the device, and it encrypts it internally and writes ciphertext to flash.) This mode is not secure, it’s possible for an attacker to write their own chosen plaintext to flash.

Release mode should always be selected for production or manufacturing. Once enabled it’s no longer possible for the device in ROM Download Mode to use the flash encryption hardware.

When EFUSEVIRTUAL is enabled, SECURE_FLASH_ENCRYPTION_MODE_RELEASE is not available. For CI tests we use IDF_CI_BUILD to bypass it (“export IDF_CI_BUILD=1”). We do not recommend bypassing it for other purposes.

Refer to the Flash Encryption section of the ESP-IDF Programmer’s Guide for details.

Available options:

- Development (NOT SECURE) (CONFIG_SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT)
- Release (CONFIG_SECURE_FLASH_ENCRYPTION_MODE_RELEASE)

**Potentially insecure options** Contains:

- CONFIG_SECURE_BOOT_V2_ALLOW_EFUSE_RD_DIS
- CONFIG_SECURE_BOOT_ALLOW_SHORT_APP_PARTITION
- CONFIG_SECURE_BOOT_ALLOW_JTAG
- CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC
- CONFIG_SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE
- CONFIG_SECURE_BOOT_ALLOW_UNUSED_DIGEST_SLOTS
- CONFIG_SECURE_FLASH_REQUIRE_ALREADY_ENABLED
- CONFIG_SECURE_FLASH_SKIP_WRITE_PROTECTION_CACHE
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.

**Default value:**
- No (disabled) if \texttt{CONFIG\_SECURE\_BOOT\_INSECURE} || \texttt{CONFIG\_SECURE\_FLASH\_ENCRYPTION\_MODE\_DEVELOPMENT}

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 \texttt{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 \texttt{EFUSE\_RD\_DIS} efuse field after Secure Boot is enabled. This may allow an attacker to read-protect the BLK2 efuse (for ESP32) and BLOCK4-BLOCK10 (i.e. BLOCK\_KEY0-BLOCK\_KEY5)(for other chips) holding the public key digest, causing an immediate denial of service and possibly allowing an additional fault injection attack to bypass the signature protection.

NOTE: Once a BLOCK is read-protected, the application will read all zeros from that block

NOTE: If ‘UART ROM download mode (Permanently disabled (recommended))’ or ‘UART ROM download mode (Permanently switch to Secure mode (recommended))’ is set, then it is __NOT__ possible to read/write efuses using espefuse.py utility. However, efuse can be read/written from the application

CONFIG_SECURE_BOOT_ALLOW_UNUSED_DIGEST_SLOTS
Leave unused digest slots available (not revoke)

*Found in: Security features > Potentially insecure options*

If not set (default), during startup in the app all unused digest slots will be revoked. To revoke unused slot will be called \texttt{esp\_efuse\_set\_digest\_revoke(num\_digest)} for each digest. Revoking unused digest slots makes ensures that no trusted keys can be added later by an attacker. If set, it means that you have a plan to use unused digests slots later.
Default value:
• No (disabled) if `CONFIG_SECURE_BOOT_INSECURE`

`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.

Default value:
• No (disabled) if `CONFIG_SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT`

`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.

Default value:
• No (disabled) if `CONFIG_SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT`

`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.

Default value:
• No (disabled) if `CONFIG_SECURE_FLASH_ENCRYPTION_MODE_DEVELOPMENT`

`CONFIG_SECURE_FLASH_SKIP_WRITE_PROTECTION_CACHE`

Skip write-protection of DIS_CACHE (DIS_ICACHE, DIS_DCACHE)

*Found in: Security features > Potentially insecure options*

If not set (default), on the first boot the bootloader will burn the write-protection of DIS_CACHE (for ESP32) or DIS_ICACHE/DIS_DCACHE (for other chips) eFuse when Flash Encryption is enabled. Write protection for cache disable efuse prevents the chip from being blocked if it is set by accident. App and bootloader use cache so disabling it makes the chip useless for IDF.

Due to other eFuses are linked with the same write protection bit (see the list below) then write-protection will not be done if these `SECURE_FLASH_UART_BOOTLOADER_ALLOW_ENC`, `SECURE_BOOT_ALLOW_JTAG` or `SECURE_FLASH_UART_BOOTLOADER_ALLOW_CACHE` options are selected to give a chance to turn on the chip into the release mode later.

List of eFuses with the same write protection bit: ESP32: MAC, MAC_CRC, DISABLE_APP_CPU, DISABLE_BT, DIS_CACHE, VOL_LEVEL_HP_INV.
ESP32-C3: DIS_ICACHE, DIS_USB_JTAG, DIS_DOWNLOAD_ICACHE, DIS_USB_SERIAL_JTAG, DIS_FORCE_DOWNLOAD, DIS_TWAI, JTAG_SEL_ENABLE, DIS_PAD_JTAG, DIS_DOWNLOAD_MANUAL_ENCRYPT.

ESP32-C6: SWAP_UART_SDIO_EN, DIS_ICACHE, DIS_USB_JTAG, DIS_DOWNLOAD_ICACHE, DIS_USB_SERIAL_JTAG, DIS_FORCE_DOWNLOAD, DIS_TWAI, JTAG_SEL_ENABLE, DIS_PAD_JTAG, DIS_DOWNLOAD_MANUAL_ENCRYPT.

ESP32-H2: DIS_ICACHE, DIS_USB_JTAG, POWERGLITCH_EN, DIS_FORCE_DOWNLOAD, SPI_DOWNLOAD_MSPI_DIS, DIS_TWAI, JTAG_SEL_ENABLE, DIS_PAD_JTAG, DIS_DOWNLOAD_MANUAL_ENCRYPT.

ESP32-S2: DIS_ICACHE, DIS_DCACHE, DIS_DOWNLOAD_ICACHE, DIS_DOWNLOAD_DCACHE, DIS_FORCE_DOWNLOAD, DIS_USB, DIS_TWAI, DIS_BOOT_REMAP, SOFT_DIS_JTAG, HARD_DIS_JTAG, DIS_DOWNLOAD_MANUAL_ENCRYPT.

ESP32-S3: DIS_ICACHE, DIS_DCACHE, DIS_DOWNLOAD_ICACHE, DIS_DOWNLOAD_DCACHE, DIS_FORCE_DOWNLOAD, DIS_USB_OTG, DIS_TWAI, DIS_APP_CPU, DIS_PAD_JTAG, DIS_DOWNLOAD_MANUAL_ENCRYPT, DIS_USB_JTAG, DIS_USB_SERIAL_JTAG, STRAP_JTAG_SEL, USB_PHY_SEL.

CONFIG_SECURE_FLASH_CHECK_ENC_EN_IN_APP

Check Flash Encryption enabled on app startup

Found in: Security features

If set (default), in an app during startup code, there is a check of the flash encryption eFuse bit is on (as the bootloader should already have set it). The app requires this bit is on to continue work otherwise abort.

If not set, the app does not care if the flash encryption eFuse bit is set or not.

Default value:
• Yes (enabled) if CONFIG_SECURE_FLASH_ENC_ENABLED

CONFIG_SECURE_UART_ROM_DL_MODE

UART ROM download mode

Found in: Security features

Available options:

• UART ROM download mode (Permanently disabled (recommended)) (CONFIG_SECURE_DISABLE_ROM_DL_MODE)
  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.

• UART ROM download mode (Permanently switch to Secure mode (recommended)) (CONFIG_SECURE_ENABLE_SECURE_ROM_DL_MODE)
  If set, during startup the app will burn an eFuse bit to permanently switch the UART ROM Download Mode into a separate Secure Download mode. This option can only work if Download Mode is not already disabled by eFuse.
Secure Download mode limits the use of Download Mode functions to update SPI config, changing baud rate, basic flash write and a command to return a summary of currently enabled security features (get_security_info).

Secure Download mode is not compatible with the esptool.py flasher stub feature, esp-efuse.py, read/writing memory or registers, encrypted download, or any other features that interact with unsupported Download Mode commands.

Secure Download mode should be enabled in any application where Flash Encryption and/or Secure Boot is enabled. Disabling this option does not immediately cancel the benefits of the security features, but it increases the potential “attack surface” for an attacker to try and bypass them with a successful physical attack.

It is also possible to enable secure download mode at runtime by calling esp_efuse_enable_rom_secure_download_mode()

Note: Secure Download mode is not available for ESP32 (includes revisions till ECO3).

- UART ROM download mode (Enabled (not recommended)) (CONFIG_SECURE_INSECURE_ALLOW_DL_MODE)
  This is a potentially insecure option. Enabling this option will allow the full UART download mode to stay enabled. This option SHOULD NOT BE ENABLED for production use cases.

**Application manager**

Contains:

- CONFIG_APP_EXCLUDE_PROJECT_NAME_VAR
- CONFIG_APP_EXCLUDE_PROJECT_VER_VAR
- CONFIG_APP_PROJECT_VER_FROM_CONFIG
- CONFIG_APP_RETRIEVE_LEN_ELF_SHA
- CONFIG_APP_COMPILE_TIME_DATE

**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.

*Default value:*

- No (disabled)

**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.

*Default value:*
Chapter 2. API Reference

- No (disabled)

**CONFIG_APP_PROJECT_VER_FROM_CONFIG**

Get the project version from Kconfig

*Found in: Application manager*

If this is enabled, then config item APP_PROJECT_VER will be used for the variable PROJECT_VER. Other ways to set PROJECT_VER will be ignored.

**Default value:**
- No (disabled)

**CONFIG_APP_PROJECT_VER**

Project version

*Found in: Application manager > CONFIG_APP_PROJECT_VER_FROM_CONFIG*

**Default value:**
- 1 if CONFIG_APP_PROJECT_VER_FROM_CONFIG

**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 the embedded APP ELF SHA-256 hash value from flash and convert it into a string and store it in a RAM buffer. This ensures the panic handler and core dump will be able to print this string even when cache is disabled. The size of the buffer is APP_RETRIEVE_LEN_ELF_SHA plus the null terminator. Changing this value will change the size of this buffer, in bytes.

**Range:**
- from 8 to 64

**Default value:**
- 9

**Boot ROM Behavior**

Contains:

- **CONFIG_BOOT_ROM_LOG_SCHEME**

**CONFIGBOOT_ROM_LOG_SCHEME**

Permanently change Boot ROM output

*Found in: Boot ROM Behavior*

Controls the Boot ROM log behavior. The rom log behavior can only be changed for once, specific eFuse bit(s) will be burned at app boot stage.

**Available options:**

- Always Log (CONFIG_BOOT_ROM_LOG_ALWAYS_ON)
  - Always print ROM logs, this is the default behavior.
- Permanently disable logging (CONFIG_BOOT_ROM_LOG_ALWAYS_OFF)
  - Don’t print ROM logs.
• Log on GPIO High (CONFIG_BOOT_ROM_LOG_ON_GPIO_HIGH)
  Print ROM logs when GPIO level is high during start up. The GPIO number is chip
dependent, e.g. on ESP32-S2, the control GPIO is GPIO46.
• Log on GPIO Low (CONFIG_BOOT_ROM_LOG_ON_GPIO_LOW)
  Print ROM logs when GPIO level is low during start up. The GPIO number is chip
dependent, e.g. on ESP32-S2, the control GPIO is GPIO46.

Serial flasher config

Contains:

• CONFIG_ESPTOOLPY_AFTER
• CONFIG_ESPTOOLPY_BEFORE
• CONFIG_ESPTOOLPY_HEADER_FLASHSIZE_UPDATE
• CONFIG_ESPTOOLPY_NO_STUB
• CONFIG_ESPTOOLPY_FLASH_SAMPLE_MODE
• CONFIG_ESPTOOLPY_FLASHSIZE
• CONFIG_ESPTOOLPY_FLASHMODE
• CONFIG_ESPTOOLPY_FLASHFREQ

CONFIG_ESPTOOLPY_NO_STUB

Disable download stub

Found in: Serial flasher config

The flasher tool sends a precompiled download stub first by default. That stub allows things like com-
pressed downloads and more. Usually you should not need to disable that feature

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 (CONFIG_ESPTOOLPY_FLASHMODE_QIO)
• QOUT (CONFIG_ESPTOOLPY_FLASHMODE_QOUT)
• DIO (CONFIG_ESPTOOLPY_FLASHMODE_DIO)
• DOUT (CONFIG_ESPTOOLPY_FLASHMODE_DOUT)
• OPI (CONFIG_ESPTOOLPY_FLASHMODE_OPI)

CONFIG_ESPTOOLPY_FLASHSAMPLE_MODE

Flash Sampling Mode

Found in: Serial flasher config

Available options:

• STR Mode (CONFIG_ESPTOOLPY_FLASHSAMPLE_MODE_STR)
• DTR Mode (CONFIG_ESPTOOLPY_FLASHSAMPLE_MODE_DTR)
**CONFIG_ESPTOOLPY_FLASHFREQ**

Flash SPI speed

*Found in: Serial flasher config*

Available options:

- **120 MHz** (READ DOCS FIRST) (CONFIG_ESPTOOLPY_FLASHFREQ_120M)
  - Optional feature for QSPI Flash. Read docs and enable CONFIG_SPI_FLASH_HPM_ENA first!
  - Flash 120 MHz SDR mode is stable.
  - Flash 120 MHz DDR mode is an experimental feature, it works when the temperature is stable.

  **Risks:** If your chip powers on at a certain temperature, then after the temperature increases or decreases by approximately 20 Celsius degrees (depending on the chip), the program will crash randomly.

- **80 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_80M)
- **64 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_64M)
- **60 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_60M)
- **48 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_48M)
- **40 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_40M)
- **32 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_32M)
- **30 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_30M)
- **26 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_26M)
- **24 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_24M)
- **20 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_20M)
- **16 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_16M)
- **15 MHz** (CONFIG_ESPTOOLPY_FLASHFREQ_15M)

**CONFIG_ESPTOOLPY_FLASHSIZE**

Flash size

*Found in: Serial flasher config*

SPI flash size, in megabytes

Available options:

- **1 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_1MB)
- **2 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_2MB)
- **4 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_4MB)
- **8 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_8MB)
- **16 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_16MB)
- **32 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_32MB)
- **64 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_64MB)
- **128 MB** (CONFIG_ESPTOOLPY_FLASHSIZE_128MB)

**CONFIG_ESPTOOLPY_HEADER_FLASHSIZE_UPDATE**

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.

Enabling this option turns off the image protection against corruption by a SHA256 digest. Updating the bootloader image before flashing would invalidate the digest.
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 (CONFIG_ESPTOOLPY_BEFORE_RESET)
- No reset (CONFIG_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 (CONFIG_ESPTOOLPY_AFTER_RESET)
- Stay in bootloader (CONFIG_ESPTOOLPY_AFTER_NORESET)

Partition Table

Contains:

- CONFIG_PARTITION_TABLE_CUSTOM_FILENAME
- CONFIG_PARTITION_TABLE_MD5
- CONFIG_PARTITION_TABLE_OFFSET
- CONFIG_PARTITION_TABLE_TYPE

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. These are mostly intended for example and development use, it’s expect that for production use you will copy one of these CSV files and create a custom partition CSV for your application.

Available options:

- Single factory app, no OTA (CONFIG_PARTITION_TABLE_SINGLE_APP)
  This is the default partition table, designed to fit into a 2MB or larger flash with a single 1MB app partition.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp.csv
This partition table is not suitable for an app that needs OTA (over the air update) capability.

- Single factory app (large), no OTA (CONFIG_PARTITION_TABLE_SINGLE_APP_LARGE)
  This is a variation of the default partition table, that expands the 1MB app partition size to 1.5MB to fit more code.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp_large.csv
  This partition table is not suitable for an app that needs OTA (over the air update) capability.

- Factory app, two OTA definitions (CONFIG_PARTITION_TABLE_TWO_OTA)
  This is a basic OTA-enabled partition table with a factory app partition plus two OTA app partitions. All are 1MB, so this partition table requires 4MB or larger flash size.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_two_ota.csv

- Custom partition table CSV (CONFIG_PARTITION_TABLE_CUSTOM)
  Specify the path to the partition table CSV to use for your project.
  Consult the Partition Table section in the ESP-IDF Programmers Guide for more information.

- Single factory app, no OTA, encrypted NVS (CONFIG_PARTITION_TABLE_SINGLE_APP_ENCRYPTED_NV)
  This is a variation of the default “Single factory app, no OTA” partition table that supports encrypted NVS when using flash encryption. See the Flash Encryption section in the ESP-IDF Programmers Guide for more information.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp_encr_nvs.csv

- Single factory app (large), no OTA, encrypted NVS (CONFIG_PARTITION_TABLE_SINGLE_APP_LARGE_ENC_NV)
  This is a variation of the “Single factory app (large), no OTA” partition table that supports encrypted NVS when using flash encryption. See the Flash Encryption section in the ESP-IDF Programmers Guide for more information.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_singleapp_large_encr_nvs.csv

- Factory app, two OTA definitions, encrypted NVS (CONFIG_PARTITION_TABLE_TWO_OTA_ENCRYPTED_NV)
  This is a variation of the “Factory app, two OTA definitions” partition table that supports encrypted NVS when using flash encryption. See the Flash Encryption section in the ESP-IDF Programmers Guide for more information.
  The corresponding CSV file in the IDF directory is components/partition_table/partitions_two_ota_encr_nvs.csv

**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.

**Default value:**

- “partitions.csv”

**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.

**Default value:**
- “0x8000”

**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.

**Default value:**
- Yes (enabled)

**Compiler options**

Contains:

- `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL`
- `CONFIG_COMPILER_FLOAT_LIB_FROM`
- `CONFIG_COMPILER_RT_LIB`
- `CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT`
- `CONFIG_COMPILER_DISABLE_GCC12_WARNINGS`
- `CONFIG_COMPILER_DISABLE_GCC13_WARNINGS`
- `CONFIG_COMPILER_DUMPRTL_FILES`
- `CONFIG_COMPILER_WARN_WRITE_STRINGS`
- `CONFIG_COMPILER_CXX_EXCEPTIONS`
- `CONFIG_COMPILER_CXX_RTTI`
- `CONFIG_COMPILER_OPTIMIZATION`
- `CONFIG_COMPILER_HIDE_PATHS_MACROS`
- `CONFIG_COMPILER_STACK_CHECK_MODE`

**CONFIG_COMPILER_OPTIMIZATION**

Optimization Level

*Found in: Compiler options*

This option sets compiler optimization level (gcc -O argument) for the app.

- The “Debug” 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) (CONFIG_COMPILER_OPTIMIZATION_DEBUG)
- Optimize for size (-Os) (CONFIG_COMPILER_OPTIMIZATION_SIZE)
- Optimize for performance (-O2) (CONFIG_COMPILER_OPTIMIZATION_PERF)
- Debug without optimization (-O0) (CONFIG_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 (CONFIG_COMPILER_OPTIMIZATION_ASSERTIONS_ENABLE)
  Enable assertions. Assertion content and line number will be printed on failure.
- Silent (saves code size) (CONFIG_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) (CONFIG_COMPILER_OPTIMIZATION_ASSERTIONS_DISABLE)
  If assertions are disabled, -DNDEBUG is added to CPPFLAGS.

**CONFIG_COMPILER_FLOAT_LIB_FROM**

Compiler float lib source

*Found in: Compiler options*

In the soft-fp part of libgcc, riscv version is written in C, and handles all edge cases in IEEE754, which makes it larger and performance is slow.

RVfplib is an optimized RISC-V library for FP arithmetic on 32-bit integer processors, for single and double-precision FP. RVfplib is “fast”, but it has a few exceptions from IEEE 754 compliance.

Available options:

- libgcc (CONFIG_COMPILER_FLOAT_LIB_FROM_GCCLIB)
- librvfp (CONFIG_COMPILER_FLOAT_LIB_FROM_RVFPLIB)

**CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT**

Disable messages in ESP_RETURN_ON_* and ESP_EXIT_ON_* macros

*Found in: Compiler options*

If enabled, the error messages will be discarded in following check macros: -ESP_RETURN_ON_ERROR - ESP_EXIT_ON_ERROR - ESP_RETURN_ON_FALSE - ESP_EXIT_ON_FALSE
**Default value:**
- No (disabled)

**CONFIG_COMPILER_HIDE_PATHS_MACROS**

Replace ESP-IDF and project paths in binaries

_FOUND in: Compiler options_

When expanding the `__FILE__` and `__BASE_FILE__` macros, replace paths inside ESP-IDF with paths relative to the placeholder string “IDF”, and convert paths inside the project directory to relative paths.

This allows building the project with assertions or other code that embeds file paths, without the binary containing the exact path to the IDF or project directories.

This option passes `-fmacro-prefix-map` options to the GCC command line. To replace additional paths in your binaries, modify the project CMakeLists.txt file to pass custom `-fmacro-prefix-map` or `-ffile-prefix-map` arguments.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- No (disabled)

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.

**Default value:**
- 0 if `CONFIG_COMPILER_CXX_EXCEPTIONS`

**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.

**Default value:**
- No (disabled)
**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

The performance impact includes increasing the amount of stack memory required for each task.

Available options:

- None (CONFIG_COMPILER_STACK_CHECK_MODE_NONE)
- Normal (CONFIG_COMPILER_STACK_CHECK_MODE_NORM)
- Strong (CONFIG_COMPILER_STACK_CHECK_MODE_STRONG)
- Overall (CONFIG_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 *`.

**Default value:**

- No (disabled)

**CONFIG_COMPILER_DISABLE_GCC12_WARNINGS**

Disable new warnings introduced in GCC 12

*Found in: Compiler options*

Enable this option if use GCC 12 or newer, and want to disable warnings which don’t appear with GCC 11.

**Default value:**

- No (disabled)
Chapter 2. API Reference

**CONFIG_COMPILER_DISABLE_GCC13_WARNINGS**

Disable new warnings introduced in GCC 13

*Found in:* Compiler options

Enable this option if use GCC 13 or newer, and want to disable warnings which don’t appear with GCC 12.

*Default value:*
- No (disabled)

**CONFIG_COMPILER_DUMPRTLFILES**

Dump RTL files during compilation

*Found in:* Compiler options

If enabled, RTL files will be produced during compilation. These files can be used by other tools, for example to calculate call graphs.

**CONFIG_COMPILER_RTLIB**

Compiler runtime library

*Found in:* Compiler options

Select runtime library to be used by compiler. - GCC toolchain supports libgcc only. - Clang allows to choose between libgcc or libclang_rt. - For host builds ( “linux” target), uses the default library.

*Available options:*
- libgcc (CONFIG_COMPILER_RTLIB_GCCLIB)
- libclang_rt (CONFIG_COMPILER_RTLIB_CLANGRT)
- Host (CONFIG_COMPILER_RTLIB_HOST)

**Component config**

Contains:
- ADC and ADC Calibration
- Application Level Tracing
- Bluetooth
- Common ESP-related
- Core dump
- Driver Configurations
- eFuse Bit Manager
- CONFIG_BLE_MESH
- ESP HTTP client
- ESP HTTPS OTA
- ESP HTTPS server
- ESP NETIF Adapter
- ESP PSRAM
- ESP Ringbuf
- ESP System Settings
- ESP-Driver:GPIO Configurations
- ESP-Driver:GPTimer Configurations
- ESP-Driver:MCPWM Configurations
- ESP-Driver:PCNT Configurations
- ESP-Driver:SPI Configurations
- ESP-MQTT Configurations
• ESP-TLS
• Ethernet
• Event Loop Library
• FAT Filesystem support
• FreeRTOS
• GDB Sub
• Hardware Abstraction Layer (HAL) and Low Level (LL)
• Hardware Settings
• Heap memory debugging
• High resolution timer (esp_timer)
• HTTP Server
• IEEE 802.15.4
• IPC (Inter-Processor Call)
• LCD and Touch Panel
• Log output
• LWIP
• Main Flash configuration
• mbedtls
• Newlib
• NVS
• NVS Security Provider
• OpenThread
• Partition API Configuration
• PHY
• Power Management
• Protocomm
• PThreads
• SoC Settings
• SPI Flash driver
• SPIFFS Configuration
• TCP Transport
• Ultra Low Power (ULP) Co-processor
• Unity unit testing library
• USB-OTG
• Virtual file system
• Wear Levelling
• Wi-Fi
• Wi-Fi Provisioning Manager
• Wireless Coexistence

**Application Level Tracing**
Contains:

• CONFIG_APPTRACE_DESTINATION1
• CONFIG_APPTRACE_DESTINATION2
• FreeRTOS SystemView Tracing
• CONFIG_APPTRACE_GCOV_ENABLE
• CONFIG_APPTRACE_BUF_SIZE
• CONFIG_APPTRACE_PENDING_DATA_SIZE_MAX
• CONFIG_APPTRACE_POSTMORTEM_FLUSH_THRESH
• CONFIG_APPTRACE_ONPANIC_HOST_FLUSH_TMO
• CONFIG_APPTRACE_UART_BAUDRATE
• CONFIG_APPTRACE_UART_RX_GPIO
• CONFIG_APPTRACE_UART_RX_BUFF_SIZE
• CONFIG_APPTRACE_UART_TX_MSG_SIZE
• CONFIG_APPTRACE_UART_TX_GPIO
• CONFIG_APPTRACE_UART_TX_BUFF_SIZE
Chapter 2. API Reference

CONFIG_APPTRACE_DESTINATION1

Data Destination 1

*Found in: Component config > Application Level Tracing*

Select destination for application trace: JTAG or none (to disable).

Available options:

- JTAG (CONFIG_APPTRACE_DEST_JTAG)
- None (CONFIG_APPTRACE_DEST_NONE)

CONFIG_APPTRACE_DESTINATION2

Data Destination 2

*Found in: Component config > Application Level Tracing*

Select destination for application trace: UART(XX) or none (to disable).

Available options:

- UART0 (CONFIG_APPTRACE_DEST_UART0)
- UART1 (CONFIG_APPTRACE_DEST_UART1)
- UART2 (CONFIG_APPTRACE_DEST_UART2)
- USB_CDC (CONFIG_APPTRACE_DEST_USB_CDC)
- None (CONFIG_APPTRACE_DEST_UART_NONE)

CONFIG_APPTRACE_UART_TX_GPIO

UART TX on GPIO#

*Found in: Component config > Application Level Tracing*

This GPIO is used for UART TX pin.

CONFIG_APPTRACE_UART_RX_GPIO

UART RX on GPIO#

*Found in: Component config > Application Level Tracing*

This GPIO is used for UART RX pin.

CONFIG_APPTRACE_UART_BAUDRATE

UART baud rate

*Found in: Component config > Application Level Tracing*

This baud rate is used for UART.

The app’s maximum baud rate depends on the UART clock source. If Power Management is disabled, the UART clock source is the APB clock and all baud rates in the available range will be sufficiently accurate. If Power Management is enabled, REF_TICK clock source is used so the baud rate is divided from 1MHz. Baud rates above 1Mbps are not possible and values between 500Kbps and 1Mbps may not be accurate.
Chapter 2. API Reference

**CONFIG_APPTRACE_UART_RX_BUFF_SIZE**

UART RX ring buffer size

*Found in: Component config > Application Level Tracing*

Size of the UART input ring buffer. This size related to the baudrate, system tick frequency and amount of data to transfer. The data placed to this buffer before sent out to the interface.

**CONFIG_APPTRACE_UART_TX_BUFF_SIZE**

UART TX ring buffer size

*Found in: Component config > Application Level Tracing*

Size of the UART output ring buffer. This size related to the baudrate, system tick frequency and amount of data to transfer.

**CONFIG_APPTRACE_UART_TX_MSG_SIZE**

UART TX message size

*Found in: Component config > Application Level Tracing*

Maximum size of the single message to transfer.

**CONFIG_APPTRACE_UART_TASK_PRIO**

UART Task Priority

*Found in: Component config > Application Level Tracing*

UART task priority. In case of high events rate, this parameter could be changed up to (config-MAX_PRIORITIES-1).

**Range:**
- from 1 to 32

**Default value:**
- 1

**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_BUF_SIZE**

Size of the apptrace buffer

*Found in: Component config > Application Level Tracing*

Size of the memory buffer for trace data 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_APPTRACE_SV_CPU`
- `CONFIG_APPTRACE_SV_EVT_ISR_ENTER_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_ISR_EXIT_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_ISR_TO_SCHED_ENABLE`
- `CONFIG_APPTRACE_SV_MAX_TASKS`
- `CONFIG_APPTRACE_SV_EVT_IDLE_ENABLE`
- `CONFIG_APPTRACE_SV_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TASK_CREATE_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TASK_START_EXEC_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TASK_START_READY_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TASK_STOP_EXEC_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TASK_STOPREADY_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TIMER_ENTER_ENABLE`
- `CONFIG_APPTRACE_SV_EVT_TIMER_EXIT_ENABLE`
- `CONFIG_APPTRACE_SV_TS_SOURCE`
- `CONFIG_APPTRACE_SV_EVT_OVERFLOW_ENABLE`
- `CONFIG_APPTRACE_SV_BUF_WAIT_TMO`

**CONFIG_APPTRACE_SV_ENABLE**

SystemView Tracing Enable

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables support for SEGGER SystemView tracing functionality.

**CONFIG_APPTRACE_SV_DEST**

SystemView destination

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing > CONFIG_APPTRACE_SV_ENABLE*

SystemView with transfer data through defined interface.

Available options:

- Data destination JTAG (CONFIG_APPTRACE_SV_DEST_JTAG)
  Send SEGGER SystemView events through JTAG interface.
- Data destination UART (CONFIG_APPTRACE_SV_DEST_UART)
  Send SEGGER SystemView events through UART interface.

**CONFIG_APPTRACE_SV_CPU**

CPU to trace

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Define the CPU to trace by SystemView.
Available options:

- CPU0 (CONFIG_APPTRACE_SV_DEST_CPU_0)
  Send SEGGER SystemView events for Pro CPU.
- CPU1 (CONFIG_APPTRACE_SV_DEST_CPU_1)
  Send SEGGER SystemView events for App CPU.

**CONFIG_APPTRACE_SV_TS_SOURCE**

Timer to use as timestamp source

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

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) (CONFIG_APPTRACE_SV_TS_SOURCE_CCOUNT)
- General Purpose Timer (Timer Group) (CONFIG_APPTRACE_SV_TS_SOURCE_GPTIMER)
- esp_timer high resolution timer (CONFIG_APPTRACE_SV_TS_SOURCE_ESP_TIMER)

**CONFIG_APPTRACE_SV_MAX_TASKS**

Maximum supported tasks

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Configures maximum supported tasks in sysview debug

**CONFIG_APPTRACE_SV_BUF_WAIT_TMO**

Trace buffer wait timeout

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Configures timeout (in us) to wait for free space in trace buffer. Set to -1 to wait forever and avoid lost events.

**CONFIG_APPTRACE_SV_EVT_OVERFLOW_ENABLE**

Trace Buffer Overflow Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Trace Buffer Overflow” event.

**CONFIG_APPTRACE_SV_EVT_ISR_ENTER_ENABLE**

ISR Enter Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “ISR Enter” event.
CONFIG_APPTRACE_SV_EVT_ISR_EXIT_ENABLE
ISR Exit Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “ISR Exit” event.

CONFIG_APPTRACE_SV_EVT_ISR_TO_SCHED_ENABLE
ISR Exit to Scheduler Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “ISR to Scheduler” event.

CONFIG_APPTRACE_SV_EVT_TASK_START_EXEC_ENABLE
Task Start Execution Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “Task Start Execution” event.

CONFIG_APPTRACE_SV_EVT_TASK_STOP_EXEC_ENABLE
Task Stop Execution Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “Task Stop Execution” event.

CONFIG_APPTRACE_SV_EVT_TASK_START_READY_ENABLE
Task Start Ready State Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “Task Start Ready State” event.

CONFIG_APPTRACE_SV_EVT_TASK_STOP_READY_ENABLE
Task Stop Ready State Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “Task Stop Ready State” event.

CONFIG_APPTRACE_SV_EVT_TASK_CREATE_ENABLE
Task Create Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “Task Create” event.

CONFIG_APPTRACE_SV_EVT_TASK_TERMINATE_ENABLE
Task Terminate Event
Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing
Enables “Task Terminate” event.
CONFIG_APPTRACE_SV_EVT_IDLE_ENABLE
System Idle Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “System Idle” event.

CONFIG_APPTRACE_SV_EVT_TIMER_ENTER_ENABLE
Timer Enter Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Timer Enter” event.

CONFIG_APPTRACE_SV_EVT_TIMER_EXIT_ENABLE
Timer Exit Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

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.

CONFIG_APPTRACE_GCOV_DUMP_TASK_STACK_SIZE
Gcov dump task stack size

*Found in: Component config > Application Level Tracing > CONFIG_APPTRACE_GCOV_ENABLE*

Configures stack size of Gcov dump task

**Default value:**
- 2048 if `CONFIG_APPTRACE_GCOV_ENABLE`

**Bluetooth**
Contains:
- *Bluedroid Options*
- *CONFIG_BT_ENABLED*
- *Controller Options*
- *NimBLE Options*
- *CONFIG_BT_RELEASE_IRAM*

CONFIG_BT_ENABLED
Bluetooth

*Found in: Component config > Bluetooth*

Select this option to enable Bluetooth and show the submenu with Bluetooth configuration choices.
**CONFIG_BT_HOST**

**Host**

*Found in: Component config > Bluetooth > CONFIG_BT_ENABLED*

This helps to choose Bluetooth host stack

Available options:

- Bluedroid - Dual-mode (CONFIG_BT_BLUEDROID_ENABLED)
  This option is recommended for classic Bluetooth or for dual-mode usecases
- NimBLE - BLE only (CONFIG_BT_NIMBLE_ENABLED)
  This option is recommended for BLE only usecases to save on memory
- Disabled (CONFIG_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).

**CONFIG_BT_CONTROLLER**

**Controller**

*Found in: Component config > Bluetooth > CONFIG_BT_ENABLED*

This helps to choose Bluetooth controller stack

Available options:

- Enabled (CONFIG_BT_CONTROLLER_ENABLED)
  This option is recommended for Bluetooth controller usecases
- Disabled (CONFIG_BT_CONTROLLER_DISABLED)
  This option is recommended for Bluetooth Host only usecases

**Bluedroid Options**  Contains:

- CONFIG_BT_BLE_HOST_QUEUE Cong Check
- CONFIG_BT_BLUEDROID_MEM_DEBUG
- CONFIG_BT_BTU_TASK_STACK_SIZE
- CONFIG_BT_BTC_TASK_STACK_SIZE
- CONFIG_BT_BLE_ENABLED
- BT Debug Log Level
- CONFIG_BT_ACL_CONNECTIONS
- CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST
- CONFIG_BT_CLASSIC_ENABLED
- CONFIG_BT_HID_ENABLED
- CONFIG_BT_STACK_NO_LOG
- CONFIG_BT_BLE_42_FEATURES_SUPPORTED
- CONFIG_BT_BLE_50_FEATURES_SUPPORTED
- CONFIG_BT_BLE_HIGH_DUTY_ADV_INTERVAL
- CONFIG_BT_MULTI_CONNECTION_ENABLE
- CONFIG_BT_BLE_FEAT_PERIODIC_ADV_SYNC_TRANSFER
- CONFIG_BT_BLUEDROID_ESP_COEX_VSC
- CONFIG_BT_BLE_FEAT_PERIODIC_ADV_ENH
- CONFIG_BT_MAX_DEVICE_NAME_LEN
- CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN
- CONFIG_BT_BLUEDROID_Pinned_TO_Core_CHOICE
- CONFIG_BT_BLE_Estab_LINK_CONN_TOUT
- CONFIG_BT_BLE_RPA_TIMEOUT
• CONFIG_BT_BLE_RPA_SUPPORTED
• CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY
• CONFIG_BT_HFP_WBS_ENABLE

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

**Default value:**
- 3072 if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

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) (CONFIG_BT_BLUEDROID_PINNED_TO_CORE_0)
- Core 1 (APP CPU) (CONFIG_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

**Default value:**
- 4096 if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_BLUEDROID_MEM_DEBUG
Bluedroid memory debug
*Found in: Component config > Bluetooth > Bluedroid Options*

Bluedroid memory debug

**Default value:**
- No (disabled) if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_BLUEDROID_ESP_COEX_VSC
Enable Espressif Vendor-specific HCI commands for coexist status configuration
*Found in: Component config > Bluetooth > Bluedroid Options*

Enable Espressif Vendor-specific HCI commands for coexist status configuration

**Default value:**
- Yes (enabled) if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED
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

**Default value:**
- No (disabled) if `CONFIG_BT_BLUEDROID_ENABLED` && ((`CONFIG_BT_CONTROLLER_ENABLED` && `SOC_BT_CLASSIC_SUPPORTED`) || `CONFIG_BT_CONTROLLER_DISABLED`) && `CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_CLASSIC_QQB_ENABLED

Host Qualification support for Classic Bluetooth

*Found in:* Component config > Bluetooth > Bluedroid Options > `CONFIG_BT_CLASSIC_ENABLED`

This enables functionalities of Host qualification for Classic Bluetooth.

**Default value:**
- No (disabled) if `CONFIG_BT_CLASSIC_ENABLED` && `CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_A2DP_ENABLE

A2DP

*Found in:* Component config > Bluetooth > Bluedroid Options > `CONFIG_BT_CLASSIC_ENABLED`

Advanced Audio Distribution Profile

**Default value:**
- No (disabled) if `CONFIG_BT_CLASSIC_ENABLED` && `CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_SPP_ENABLED

SPP

*Found in:* Component config > Bluetooth > Bluedroid Options > `CONFIG_BT_CLASSIC_ENABLED`

This enables the Serial Port Profile

**Default value:**
- No (disabled) if `CONFIG_BT_CLASSIC_ENABLED` && `CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_L2CAP_ENABLED

BT L2CAP

*Found in:* Component config > Bluetooth > Bluedroid Options > `CONFIG_BT_CLASSIC_ENABLED`

This enables the Logical Link Control and Adaptation Layer Protocol. Only supported classic bluetooth.

**Default value:**
- No (disabled) if `CONFIG_BT_CLASSIC_ENABLED` && `CONFIG_BT_BLUEDROID_ENABLED`
CONFIG_BT_HFP_ENABLE

Hands Free/Handset Profile

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED*

Hands Free Unit and Audio Gateway can be included simultaneously but they cannot run simultaneously due to internal limitations.

**Default value:**
- No (disabled) if CONFIG_BT_CLASSIC_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

**Contains:**
- CONFIG_BT_HFP_AG_ENABLE
- CONFIG_BT_HFP_AUDIO_DATA_PATH
- CONFIG_BT_HFP_CLIENT_ENABLE

CONFIG_BT_HFP_CLIENT_ENABLE

Hands Free Unit

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED > CONFIG_BT_HFP_ENABLE*

**Default value:**
- Yes (enabled) if CONFIG_BT_HFP_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_HFP_AG_ENABLE

Audio Gateway

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_CLASSIC_ENABLED > CONFIG_BT_HFP_ENABLE*

**Default value:**
- Yes (enabled) if CONFIG_BT_HFP_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

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 (CONFIG_BT_HFP_AUDIO_DATA_PATH_PCM)
- HCI (CONFIG_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.

**Default value:**
• Yes (enabled) if CONFIG_BT_HFP_AUDIO_DATA_PATH_HCI && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_HID_ENABLED

Classic BT HID

*Found in: Component config > Bluetooth > Bluedroid Options*

This enables the BT HID Host

**Default value:**
• No (disabled) if CONFIG_BT_CLASSIC_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

Contains:
• CONFIG_BT_HID_DEVICE_ENABLED
• CONFIG_BT_HID_HOST_ENABLED

CONFIG_BT_HID_HOST_ENABLED

Classic BT HID Host

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_HID_ENABLED*

This enables the BT HID Host

**Default value:**
• No (disabled) if CONFIG_BT_HID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_HIDDEVICE_ENABLED

Classic BT HID Device

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_HID_ENABLED*

This enables the BT HID Device

CONFIG_BT_BLE_ENABLED

Bluetooth Low Energy

*Found in: Component config > Bluetooth > Bluedroid Options*

This enables Bluetooth Low Energy

**Default value:**
• Yes (enabled) if CONFIG_BT_BLE_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

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

**Default value:**
• Yes (enabled) if CONFIG_BT_BLE_ENABLED && CONFIG_BT_BLUEDROID_ENABLED
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_GATTENABLE *

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

**Default value:**
- No (disabled) if CONFIG_BT_GATTENABLE && CONFIG_BT_BLEBLUEDROID_ENABLED

CONFIG_BT_BLE_BLUFI_ENABLE
Include blufi function

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTENABLE *

This option can be close when the app does not require blufi function.

**Default value:**
- No (disabled) if CONFIG_BT_GATTENABLE && CONFIG_BT_BLEBLUEDROID_ENABLED

CONFIG_BTGatt_MAX_SR_PROFILES
Max GATT Server Profiles

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTENABLE *

Maximum GATT Server Profiles Count

**Range:**
- from 1 to 32 if CONFIG_BT_GATTENABLE && CONFIG_BT_BLEBLUEDROID_ENABLED && CONFIG_BT_BLEBLUEDROID_ENABLED

**Default value:**
- 8 if CONFIG_BT_GATTENABLE && CONFIG_BT_BLEBLUEDROID_ENABLED && CONFIG_BT_BLEBLUEDROID_ENABLED

CONFIG_BTGatt_MAX_SR_ATTRIBUTES
Max GATT Service Attributes

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTENABLE *

Maximum GATT Service Attributes Count

**Range:**
- from 1 to 500 if CONFIG_BT_GATTENABLE && CONFIG_BT_BLEBLUEDROID_ENABLED && CONFIG_BT_BLEBLUEDROID_ENABLED

**Default value:**
- 100 if CONFIG_BT_GATTENABLE && CONFIG_BT_BLEBLUEDROID_ENABLED && CONFIG_BT_BLEBLUEDROID_ENABLED

CONFIG_BT_GATTS_SEND_SERVICE_CHANGE_MODE
GATTS Service Change Mode

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTENABLE *
Chapter 2. API Reference

Service change indication mode for GATT Server.

Available options:

- **GATTS manually send service change indication** (CONFIG_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** (CONFIG_BT_GATTS_SEND_SERVICE_CHANGE_AUTO)
  Let Bluedroid handle the service change indication internally

**CONFIG_BT_GATTS_ROBUST_CACHING_ENABLED**

Enable Robust Caching on Server Side

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE*

This option enable gatt robust caching feature on server

**Default value:**
- No (disabled) if `CONFIG_BT_GATTS_ENABLE` && `CONFIG_BT_BLUEDROID_ENABLED`

**CONFIG_BT_GATTS_DEVICE_NAME_WRITABLE**

Allow to write device name by GATT clients

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE*

Enabling this option allows remote GATT clients to write device name

**Default value:**
- No (disabled) if `CONFIG_BT_GATTS_ENABLE` && `CONFIG_BT_BLUEDROID_ENABLED`

**CONFIG_BT_GATTS_APPEARANCE_WRITABLE**

Allow to write appearance by GATT clients

*Found in: Component config > Bluetooth > Bluedroid Options > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTS_ENABLE*

Enabling this option allows remote GATT clients to write appearance

**Default value:**
- No (disabled) if `CONFIG_BT_GATTS_ENABLE` && `CONFIG_BT_BLUEDROID_ENABLED`

**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

**Default value:**
- Yes (enabled) if `CONFIG_BT_BLE_ENABLED` && `CONFIG_BT_BLUEDROID_ENABLED`
CONFIG_BT_GATTC_MAX_CACHE_CHAR
Max gattc cache characteristic for discover

*Found in:* Component config > Bluetooth > Bluez > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

Maximum GATT cache characteristic count

**Range:**
- from 1 to 500 if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

**Default value:**
- 40 if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_NOTIF_REG_MAX
Max gattc notify(indication) register number

*Found in:* Component config > Bluetooth > Bluez > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

Maximum GATT notify(indication) register number

**Range:**
- from 1 to 64 if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

**Default value:**
- 5 if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_CACHE_NVS_FLASH
Save gattc cache data to nvs flash

*Found in:* Component config > Bluetooth > Bluez > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

This select can save gattc cache data to nvs flash

**Default value:**
- No (disabled) if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_GATTC_CONNECT_RETRY_COUNT
The number of attempts to reconnect if the connection establishment failed

*Found in:* Component config > Bluetooth > Bluez > CONFIG_BT_BLE_ENABLED > CONFIG_BT_GATTC_ENABLE

The number of attempts to reconnect if the connection establishment failed

**Range:**
- from 0 to 7 if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

**Default value:**
- 3 if CONFIG_BT_GATTC_ENABLE && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_BLE_SMP_ENABLE
Include BLE security module(SMP)

*Found in:* Component config > Bluetooth > Bluez > CONFIG_BT_BLE_ENABLED

This option can be close when the app not used the ble security connect.

**Default value:**
- Yes (enabled) if CONFIG_BT_BLE_ENABLED && CONFIG_BT_BLUEDROID_ENABLED
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.

**Default value:**
- No (disabled) if CONFIG_BT_BLE_SMP_ENABLE && CONFIG_BT_BLE_ENABLED

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

**Default value:**
- No (disabled) if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

BT DEBUG LOG LEVEL  Contains:

- CONFIG_BT_LOG_A2D_TRACE_LEVEL
- CONFIG_BT_LOG_APL_TRACE_LEVEL
- CONFIG_BT_LOG_AVCT_TRACE_LEVEL
- CONFIG_BT_LOG_AVDT_TRACE_LEVEL
- CONFIG_BT_LOG_AVRC_TRACE_LEVEL
- CONFIG_BT_LOG_BFC_TRACE_LEVEL
- CONFIG_BT_LOG_BNEP_TRACE_LEVEL
- CONFIG_BT_LOG_BTIF_TRACE_LEVEL
- CONFIG_BT_LOG_BTM_TRACE_LEVEL
- CONFIG_BT_LOG_GAP_TRACE_LEVEL
- CONFIG_BT_LOG_GATT_TRACE_LEVEL
- CONFIG_BT_LOG_HCI_TRACE_LEVEL
- CONFIG_BT_LOG_HID_TRACE_LEVEL
- CONFIG_BT_LOG_L2CAP_TRACE_LEVEL
- CONFIG_BT_LOG_MCA_TRACE_LEVEL
- CONFIG_BT_LOG_OSI_TRACE_LEVEL
- CONFIG_BT_LOG_PAN_TRACE_LEVEL
- CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL
- CONFIG_BT_LOG_SDP_TRACE_LEVEL
- CONFIG_BT_LOG_SMP_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 (CONFIG_BT_LOG_HCI_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_HCI_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_HCI_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_HCI_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_HCI_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_HCI_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_BTM_TRACE_LEVEL_NONE)
• ERROR (CONFIG_BT_LOG_BTM_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_BTM_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_BTM_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_BTM_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_BTM_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL_NONE)
• ERROR (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL_NONE)
• ERROR (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_SDP_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_SDP_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_SDP_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_SDP_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_SDP_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_SDP_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_GAP_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_GAP_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_GAP_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_GAP_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_GAP_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_GAP_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_BNEP_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_BNEP_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_BNEP_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_BNEP_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_BNEP_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_BNEP_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_PAN_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_PAN_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_PAN_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_PAN_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_PAN_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_PAN_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_A2D_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_A2D_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_A2D_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_A2D_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_A2D_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_A2D_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_BT_LOG_A2D_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVDT_TRACE_LEVEL

AVDT layer

*Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL*

Define BT trace level for AVDT layer

Available options:

- NONE (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_BT_LOG_AVDT_TRACE_LEVEL_VERBOSE)

CONFIG_BT_LOG_AVCT_TRACE_LEVEL

AVCT layer

*Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL*

Define BT trace level for AVCT layer

Available options:
Chapter 2. API Reference

- NONE (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) None)
- ERROR (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) Error)
- WARNING (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) Warning)
- API (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) Api)
- EVENT (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) Event)
- DEBUG (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) Debug)
- VERBOSE (CONFIG_BT_LOG_AVCT_TRACE(LogLevel) Verbose)

**CONFIG_BT_LOG_AVRC_TRACE_LEVEL**

AVRC layer

*Found in: Component config > Bluetooth > Bluedroid Options > BT DEBUG LOG LEVEL*

Define BT trace level for AVRC layer

Available options:

- NONE (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) None)
- ERROR (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) Error)
- WARNING (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) Warning)
- API (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) Api)
- EVENT (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) Event)
- DEBUG (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) Debug)
- VERBOSE (CONFIG_BT_LOG_AVRC_TRACE(LogLevel) 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 (CONFIG_BT_LOG_MCA_TRACE(LogLevel) None)
- ERROR (CONFIG_BT_LOG_MCA_TRACE(LogLevel) Error)
- WARNING (CONFIG_BT_LOG_MCA_TRACE(LogLevel) Warning)
- API (CONFIG_BT_LOG_MCA_TRACE(LogLevel) Api)
- EVENT (CONFIG_BT_LOG_MCA_TRACE(LogLevel) Event)
- DEBUG (CONFIG_BT_LOG_MCA_TRACE(LogLevel) Debug)
- VERBOSE (CONFIG_BT_LOG_MCA_TRACE(LogLevel) 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 (CONFIG_BT_LOG_HID_TRACE(LogLevel) None)
- ERROR (CONFIG_BT_LOG_HID_TRACE(LogLevel) Error)
- WARNING (CONFIG_BT_LOG_HID_TRACE(LogLevel) Warning)
- API (CONFIG_BT_LOG_HID_TRACE(LogLevel) Api)
- EVENT (CONFIG_BT_LOG_HID_TRACE(LogLevel) Event)
• DEBUG (CONFIG_BT_LOG_HID_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_APPL_TRACE_LEVEL_NONE)
• ERROR (CONFIG_BT_LOG_APPL_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_APPL_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_APPL_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_APPL_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_APPL_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_GATT_TRACE_LEVEL_NONE)
• ERROR (CONFIG_BT_LOG_GATT_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_GATT_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_GATT_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_GATT_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_GATT_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_SMP_TRACE_LEVEL_NONE)
• ERROR (CONFIG_BT_LOG_SMP_TRACE_LEVEL_ERROR)
• WARNING (CONFIG_BT_LOG_SMP_TRACE_LEVEL_WARNING)
• API (CONFIG_BT_LOG_SMP_TRACE_LEVEL_API)
• EVENT (CONFIG_BT_LOG_SMP_TRACE_LEVEL_EVENT)
• DEBUG (CONFIG_BT_LOG_SMP_TRACE_LEVEL_DEBUG)
• VERBOSE (CONFIG_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 (CONFIG_BT_LOG_BTIF_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_BTIF_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_BTIF_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_BTIF_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_BTIF_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_BTIF_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_BTC_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_BTC_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_BTC_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_BTC_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_BTC_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_BTC_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_OSI_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_OSI_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_OSI_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_OSI_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_OSI_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_OSI_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_WARNING)
- API (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_API)
- EVENT (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_EVENT)
- DEBUG (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL_VERBOSE)

CONFIG_BT_ACL_CONNECTIONS

BT/BLE MAX ACL CONNECTIONS(1-9)

Found in: Component config > Bluetooth > Bluedroid Options

Maximum BT/BLE connection count. The ESP32-C3/S3 chip supports a maximum of 10 instances, including ADV, SCAN and connections. The ESP32-C3/S3 chip can connect up to 9 devices if ADV or SCAN uses only one. If ADV and SCAN are both used, the ESP32-C3/S3 chip is connected to a maximum of 8 devices. Because Bluetooth cannot reclaim used instances once ADV or SCAN is used.

Range:
- from 1 to 9 if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

Default value:
- 4 if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

CONFIG_BT_MULTI_CONNECTION_ENBALE

Enable BLE multi-conections

Found in: Component config > Bluetooth > Bluedroid Options

Enable this option if there are multiple connections

Default value:
- Yes (enabled) if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

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

Default value:
- No (disabled) if CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED

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

Default value:
• No (disabled) if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

**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.

**Default value:**

• No (disabled) if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

**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

**Default value:**

• No (disabled) if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLE_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

**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.

**Range:**

• from 1 to 60 if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

**Default value:**

• 30 if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

**CONFIG_BT_MAX_DEVICE_NAME_LEN**

length of bluetooth device name

*Found in: Component config > Bluetooth > Bluedroid Options*

Bluetooth Device name length shall be no larger than 248 octets, If the broadcast data cannot contain the complete device name, then only the shortname will be displayed, the rest parts that can’t fit in will be truncated.

**Range:**

• from 32 to 248 if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

**Default value:**

• 32 if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`
CONFIG_BT_BLE_RPA_SUPPORTED

Update RPA to Controller

*Found in: Component config > Bluetooth > Bluedroid Options*

This enables controller RPA list function. For ESP32, ESP32 only support network privacy mode. If this option is enabled, ESP32 will only accept advertising packets from peer devices that contain private address, HW will not receive the advertising packets contain identity address after IRK changed. If this option is disabled, address resolution will be performed in the host, so the functions that require controller to resolve address in the white list cannot be used. This option is disabled by default on ESP32, please enable or disable this option according to your own needs.

For other BLE chips, devices support network privacy mode and device privacy mode, users can switch the two modes according to their own needs. So this option is enabled by default.

*Default value:*

- Yes (enabled) if `CONFIG_BT_CONTROLLER_DISABLED && CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_CONTROLLER_DISABLED && CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_BLE_RPA_TIMEOUT

Timeout of resolvable private address

*Found in: Component config > Bluetooth > Bluedroid Options*

This set RPA timeout of Controller and Host. Default is 900 s (15 minutes). Range is 1 s to 1 hour (3600 s).

*Range:*

- from 1 to 3600 if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

*Default value:*

- 900 if `CONFIG_BT_BLUEDROID_ENABLED && CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_BLE_50_FEATURES_SUPPORTED

Enable BLE 5.0 features

*Found in: Component config > Bluetooth > Bluedroid Options*

This enables BLE 5.0 features, this option only support esp32c3/esp32s3 chip

*Default value:*

- Yes (enabled) if `CONFIG_BT_BLUEDROID_ENABLED && ((CONFIG_BT_CONTROLLER_ENABLED && SOC_BLE_50_SUPPORTED) || CONFIG_BT_CONTROLLER_DISABLED) && CONFIG_BT_BLUEDROID_ENABLED`

CONFIG_BT_BLE_42_FEATURES_SUPPORTED

Enable BLE 4.2 features

*Found in: Component config > Bluetooth > Bluedroid Options*

This enables BLE 4.2 features.

*Default value:*

- No (disabled) if `CONFIG_BT_BLUEDROID_ENABLED && ((CONFIG_BT_CONTROLLER_ENABLED && SOC_BLE_SUPPORTED) || CONFIG_BT_CONTROLLER_DISABLED) && CONFIG_BT_BLUEDROID_ENABLED`
**CONFIG_BT_BLE_FEAT_PERIODIC_ADV_SYNC_TRANSFER**

Enable BLE periodic advertising sync transfer feature

*Found in: Component config > Bluetooth > Bluedroid Options*

This enables BLE periodic advertising sync transfer feature

**Default value:**

- No (disabled) if `CONFIG_BT_BLE_FEAT_PERIODIC_ADV_SYNC_TRANSFER` enabled

**CONFIG_BT_BLE_FEAT_PERIODIC_ADV_ENH**

Enable periodic adv enhancements(adi support)

*Found in: Component config > Bluetooth > Bluedroid Options*

Enable the periodic advertising enhancements

**Default value:**

- No (disabled) if `CONFIG_BT_BLE_FEAT_PERIODIC_ADV_ENH` enabled

**CONFIG_BT_BLE_HIGH_DUTY_ADV_INTERVAL**

Enable BLE high duty advertising interval feature

*Found in: Component config > Bluetooth > Bluedroid Options*

This enable BLE high duty advertising interval feature

**Default value:**

- No (disabled) if `CONFIG_BT_BLE_HIGH_DUTY_ADV_INTERVAL` enabled

**NimBLE Options**

Contains:

- `CONFIG_BT_NIMBLE_SVC_GAP_DEVICE_NAME`
- `CONFIG_BT_NIMBLE_HS_STOP_TIMEOUT_MS`
- `CONFIG_BT_NIMBLE_HOST_QUEUE_CONG_CHECK`
- `CONFIG_BT_NIMBLE_WHITELIST_SIZE`
- `CONFIG_BT_NIMBLE_BLE_GATT_BLOB_TRANSFER`
- `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT`
- `CONFIG_BT_NIMBLE_ROLE_BROADCASTER`
- `CONFIG_BT_NIMBLE_ROLE_PERIPHERAL`
- `CONFIG_BT_NIMBLE_SECURITY_ENABLE`
- `CONFIG_BT_NIMBLE_BLUFI_ENABLE`
- `CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT`
- `CONFIG_BT_NIMBLE_DYNAMIC_SERVICE`
- `CONFIG_BT_NIMBLE_USE_ESP_TIMER`
- `CONFIG_BT_NIMBLE_DEBUG`
- `CONFIG_BT_NIMBLE_HS_FLOW_CTRL`
- `CONFIG_BT_NIMBLE_vs_SUPPORT`
Chapter 2. API Reference

- `CONFIG_BT_NIMBLE_OPTIMIZE_MULTI_CONN`
- `CONFIG_BT_NIMBLE_ENC_ADV_DATA`
- `CONFIG_BT_NIMBLE_SVC_GAP_APPEARANCE`
- GAP Service
- `CONFIG_BT_NIMBLE_GAP_DEVICE_NAME_MAX_LEN`
- `CONFIG_BT_NIMBLE_MAX_BONDS`
- `CONFIG_BT_NIMBLE_MAX_CCCDS`
- `CONFIG_BT_NIMBLE_MAX_CONNECTIONS`
- `CONFIG_BT_NIMBLE_L2CAP_COC_MAX_NUM`
- `CONFIG_BT_NIMBLE_GATT_MAX_PROCS`
- `CONFIG_BT_NIMBLE_MEM_ALLOC_MODE`
- Memory Settings
- `CONFIG_BT_NIMBLE_LOG_LEVEL`
- `CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE`
- `CONFIG_BT_NIMBLE_CRYPTO_STACK_MBEDTLS`
- `CONFIG_BT_NIMBLE_NVSH_PERSIST`
- `CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU`
- `CONFIG_BT_NIMBLE_RPA_TIMEOUT`
- `CONFIG_BT_NIMBLE_PINNED_TO_CORE_CHOICE`
- `CONFIG_BT_NIMBLE_TEST_THROUGHPUT_TEST`

### `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
- Internal IRAM memory wherever applicable else internal DRAM

Available options:

- Internal memory (`CONFIG_BT_NIMBLE_MEM_ALLOC_MODE_INTERNAL`)
- External SPIRAM (`CONFIG_BT_NIMBLE_MEM_ALLOC_MODE_EXTERNAL`)
- Default alloc mode (`CONFIG_BT_NIMBLE_MEM_ALLOC_MODE_DEFAULT`)
- Internal IRAM (`CONFIG_BT_NIMBLE_MEM_ALLOC_MODE_IRAM_8BIT`)
  
  Allows to use IRAM memory region as 8bit accessible region.
  
  Every unaligned (8bit or 16bit) access will result in an exception and incur penalty of certain clock cycles per unaligned read/write.

### `CONFIG_BT_NIMBLE_LOG_LEVEL`

NimBLE Host log verbosity

*Found in:* `Component config > Bluetooth > NimBLE Options`

Select NimBLE log level. Please make a note that the selected NimBLE log verbosity can not exceed the level set in “Component config > Log output > Default log verbosity”.

Available options:

- No logs (`CONFIG_BT_NIMBLE_LOG_LEVEL_NONE`)
- Error logs (`CONFIG_BT_NIMBLE_LOG_LEVEL_ERROR`)
- Warning logs (`CONFIG_BT_NIMBLE_LOG_LEVEL_WARNING`
• Info logs (CONFIG_BT_NIMBLE_LOG_LEVEL_INFO)
• Debug logs (CONFIG_BT_NIMBLE_LOG_LEVEL_DEBUG)

**CONFIG_BT_NIMBLE_MAX_CONNECTIONS**

Maximum number of concurrent connections

*Found in: Component config > Bluetooth > NimBLE Options*

Defines maximum number of concurrent BLE connections. For ESP32, user is expected to configure BTDM_CTRL_BLE_MAX_CONN from controller menu along with this option. Similarly for ESP32-C3 or ESP32-S3, user is expected to configure BT_CTRL_BLE_MAX_ACT from controller menu. For ESP32C2, ESP32C6 and ESP32H2, each connection will take about 1k DRAM.

*Range:*

• from 1 to 9 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

*Default value:*

• 3 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**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

*Default value:*

• 3 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**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

*Default value:*

• 8 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**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

*Range:*

• from 0 to 9 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

*Default value:*

• 0 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**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:
CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE

NimBLE Host task stack size

*Found in: Component config > Bluetooth > NimBLE Options*

This configures stack size of NimBLE host task

**Default value:**
- $5120$ if $\text{CONFIG_BLE_MESH} \&\& \text{CONFIG_BT_NIMBLE_ENABLED} \&\& \text{CONFIG_BT_NIMBLE_ENABLED}$
- $4096$ if $\text{CONFIG_BT_NIMBLE_ENABLED} \&\& \text{CONFIG_BT_NIMBLE_ENABLED}$

CONFIG_BT_NIMBLE_ROLE_CENTRAL

Enable BLE Central role

*Found in: Component config > Bluetooth > NimBLE Options*

Enables central role

**Default value:**
- Yes (enabled) if $\text{CONFIG_BT_NIMBLE_ENABLED} \&\& \text{CONFIG_BT_NIMBLE_ENABLED}$

CONFIG_BT_NIMBLE_ROLE_PERIPHERAL

Enable BLE Peripheral role

*Found in: Component config > Bluetooth > NimBLE Options*

Enable peripheral role

**Default value:**
- Yes (enabled) if $\text{CONFIG_BT_NIMBLE_ENABLED} \&\& \text{CONFIG_BT_NIMBLE_ENABLED}$

CONFIG_BT_NIMBLE_ROLE_BROADCASTER

Enable BLE Broadcaster role

*Found in: Component config > Bluetooth > NimBLE Options*

Enables broadcaster role

**Default value:**
- Yes (enabled) if $\text{CONFIG_BT_NIMBLE_ENABLED} \&\& \text{CONFIG_BT_NIMBLE_ENABLED}$

CONFIG_BT_NIMBLE_ROLE_OBSERVER

Enable BLE Observer role

*Found in: Component config > Bluetooth > NimBLE Options*

Enables observer role

**Default value:**
- Yes (enabled) if $\text{CONFIG_BT_NIMBLE_ENABLED} \&\& \text{CONFIG_BT_NIMBLE_ENABLED}$
**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

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_BT_NIMBLE_SECURITY_ENABLE**
Enable BLE SM feature
*Found in: Component config > Bluetooth > NimBLE Options*
Enable BLE sm feature

**Default value:**
- Yes (enabled) if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**Contains:**
- `CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_ENCRYPTION`
- `CONFIG_BT_NIMBLE_SM_LEGACY`
- `CONFIG_BT_NIMBLE_SM_SC`

**CONFIG_BT_NIMBLE_SM_LEGACY**
Security manager legacy pairing
*Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE*
Enable security manager legacy pairing

**Default value:**
- Yes (enabled) if `CONFIG_BT_NIMBLE_SECURITY_ENABLE` && `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_BT_NIMBLE_SM_SC**
Security manager secure connections (4.2)
*Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE*
Enable security manager secure connections

**Default value:**
- Yes (enabled) if `CONFIG_BT_NIMBLE_SECURITY_ENABLE` && `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_BT_NIMBLE_SM_SC_DEBUG_KEYS**
Use predefined public-private key pair
*Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE > CONFIG_BT_NIMBLE_SM_SC*
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.

**Default value:**
Chapter 2. API Reference

**CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_ENCRYPTION**

Enable LE encryption

*Found in:* Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_SECURITY_ENABLE

Enable encryption connection

**Default value:**

- Yes (enabled) if CONFIG_BT_NIMBLE_SECURITY_ENABLE && CONFIG_BT_NIMBLE_ENABLED

**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

**Default value:**

- No (disabled) if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

**CONFIG_BT_NIMBLE_DYNAMIC_SERVICE**

Enable dynamic services

*Found in:* Component config > Bluetooth > NimBLE Options

This enables user to add/remove Gatt services at runtime

**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()

**Default value:**

- “nimble” if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

**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

**Default value:**

- 31 if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

Espressif Systems

Submit Document Feedback
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()

**Default value:**
- 256 if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

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

**Default value:**
- 0 if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

**Memory Settings**

Contains:

- CONFIG_BT_NIMBLE_TRANSPORT_ACL_FROM_LL_COUNT
- CONFIG_BT_NIMBLE_TRANSPORT_EVT_DISCARD_COUNT
- CONFIG_BT_NIMBLE_MSYS_BUF_FROM_HEAP
- CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT
- CONFIG_BT_NIMBLE_MSYS_1_BLOCK_SIZE
- CONFIG_BT_NIMBLE_MSYS_2_BLOCK_COUNT
- CONFIG_BT_NIMBLE_MSYS_2_BLOCK_SIZE
- CONFIG_BT_NIMBLE_TRANSPORT_ACL_SIZE
- CONFIG_BT_NIMBLE_TRANSPORT_EVT_COUNT
- CONFIG_BT_NIMBLE_TRANSPORT_EVT_SIZE

CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT

MSYS_1 Block Count

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

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.

**Default value:**
- 24 if SOC_ESP_NIMBLE_CONTROLLER && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MSYS_1_BLOCK_SIZE

MSYS_1 Block Size

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

Dynamic memory size of block 1

**Default value:**
- 128 if SOC_ESP_NIMBLE_CONTROLLER && CONFIG_BT_NIMBLE_ENABLED
CONFIG_BT_NIMBLE_MSYS_2_BLOCK_COUNT

MSYS_2 Block Count

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

Dynamic memory count

**Default value:**

- 24 if `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_MSYS_2_BLOCK_SIZE

MSYS_2 Block Size

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

Dynamic memory size of block 2

**Default value:**

- 320 if `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_MSYS_BUF_FROM_HEAP

Get Msys Mbuf from heap

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

This option sets the source of the shared msys mbuf memory between the Host and the Controller. Allocate the memory from the heap if this option is set, from the mempool otherwise.

**Default value:**

- Yes (enabled) if `BT_LE_MSYS_INIT_IN_CONTROLLER` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_TRANSPORT_ACL_FROM_LL_COUNT

ACL Buffer count

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

The number of ACL data buffers allocated for host.

**Default value:**

- 24 if `CONFIG_BT_NIMBLE_ENABLED` & `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_TRANSPORT_ACL_SIZE

Transport ACL Buffer size

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

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)

**Default value:**

- 255 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_TRANSPORT_EVT_SIZE

Transport Event Buffer size

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

This is the size of each HCI event buffer in bytes. In case of extended advertising, packets can be fragmented. 257 bytes is the maximum size of a packet.
Default value:
- 257 if `CONFIG_BT_NIMBLE_EXT_ADV` && `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`
- 70 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_TRANSPORT_EVT_COUNT
Transport Event Buffer count

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

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

Default value:
- 30 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_TRANSPORT_EVT_DISCARD_COUNT
Discardable Transport Event Buffer count

*Found in: Component config > Bluetooth > NimBLE Options > Memory Settings*

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

Default value:
- 8 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_GATT_MAX_PROCS
Maximum number of GATT client procedures

*Found in: Component config > Bluetooth > NimBLE Options*

Maximum number of GATT client procedures that can be executed.

Default value:
- 4 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_HS_FLOW_CTRL
Enable Host Flow control

*Found in: Component config > Bluetooth > NimBLE Options*

Enable Host Flow control

Default value:
- No (disabled) if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

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

Default value:
- 1000 if `CONFIG_BT_NIMBLE_HS_FLOW_CTRL` && `CONFIG_BT_NIMBLE_ENABLED`
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

**Default value:**
- 2 if `CONFIG_BT_NIMBLE_HS_FLOW_CTRL` && `CONFIG_BT_NIMBLE_ENABLED`

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

**Default value:**
- Yes (enabled) if `CONFIG_BT_NIMBLE_HS_FLOW_CTRL` && `CONFIG_BT_NIMBLE_ENABLED`

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

**Range:**
- from 1 to 41400 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

**Default value:**
- 900 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_MESH

Enable BLE mesh functionality

*Found in: Component config > Bluetooth > NimBLE Options*

Enable BLE Mesh example present in upstream mynewt-nimble and not maintained by Espressif.

IDF maintains ESP-BLE-MESH as the official Mesh solution. Please refer to ESP-BLE-MESH guide at: `doc://esp32/api-guides/esp-ble-mesh/ble-mesh-index`

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

Contains:
- `CONFIG_BT_NIMBLE_MESH_PROVISIONER`
- `CONFIG_BT_NIMBLE_MESH_PROV`
- `CONFIG_BT_NIMBLE_MESH_GATT_PROXY`
- `CONFIG_BT_NIMBLE_MESH_FRIEND`
- `CONFIG_BT_NIMBLE_MESH_LOW_POWER`
- `CONFIG_BT_NIMBLE_MESH_PROXY`
- `CONFIG_BT_NIMBLE_MESH_RELAY`
- `CONFIG_BT_NIMBLE_MESH_DEVICE_NAME`
- `CONFIG_BT_NIMBLE_MESH_NODE_COUNT`
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

**Default value:**
- No (disabled) if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_PROV

Enable BLE mesh provisioning

*Found in:* Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable mesh provisioning

**Default value:**
- Yes (enabled) if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

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

**Default value:**
- Yes (enabled) if CONFIG_BT_NIMBLE_MESH_PROV && CONFIG_BT_NIMBLE_ENABLED

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

**Default value:**
- Yes (enabled) if CONFIG_BT_NIMBLE_MESH_PROV && CONFIG_BT_NIMBLE_ENABLED

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

**Default value:**
- Yes (enabled) if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED
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

Default value:
• No (disabled) if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

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

Default value:
• No (disabled) if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

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

Default value:
• No (disabled) if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

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

Default value:
• “nimble-mesh-node” if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_NODE_COUNT
Set mesh node count

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Defines mesh node count.

Default value:
• 1 if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MESH_PROVISIONER
Enable BLE mesh provisioner

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_MESH

Enable mesh provisioner.

Default value:
• 0 if CONFIG_BT_NIMBLE_MESH && CONFIG_BT_NIMBLE_ENABLED
CONFIG_BT_NIMBLE_CRYPTO_STACK_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.

**Default value:**
- Yes (enabled) if `CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_HS_STOP_TIMEOUT_MS

BLE host stop timeout in msec

*Found in:* Component config > Bluetooth > NimBLE Options

BLE Host stop procedure timeout in milliseconds.

**Default value:**
- 2000 if `CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT

Enable connection reattempts on connection establishment error

*Found in:* Component config > Bluetooth > NimBLE Options

Enable to make the NimBLE host to reattempt GAP connection on connection establishment failure.

**Default value:**
- Yes (enabled) if SOC_ESP_NIMBLE_CONTROLLER && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_MAX_CONN_REATTEMPT

Maximum number connection reattempts

*Found in:* Component config > Bluetooth > NimBLE Options > `CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT`

Defines maximum number of connection reattempts.

**Range:**
- from 1 to 7 if `CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT && CONFIG_BT_NIMBLE_ENABLED`

**Default value:**
- 3 if `CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT && CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Enable BLE 5 feature

*Found in:* Component config > Bluetooth > NimBLE Options

Enable BLE 5 feature

**Default value:**
- Yes (enabled) if `CONFIG_BT_NIMBLE_ENABLED && SOC_BLE_50_SUPPORTED && CONFIG_BT_NIMBLE_ENABLED`

Contains:
- `CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_2M PHY`
- `CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_CODED_PHY`
• `CONFIG_BT_NIMBLE_EXT_ADV`
• `CONFIG_BT_NIMBLE_BLE_POWER_CONTROL`
• `CONFIG_BT_NIMBLE_MAX_PERIODIC_ADVERTISER_LIST`
• `CONFIG_BT_NIMBLE_MAX_PERIODIC_SYNCS`
• `CONFIG_BT_NIMBLE_PERIODIC_ADVERTISER_ENH`

**CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_2M_PHY**

Enable 2M Phy

*Found in:* Component config > Bluetooth > NimBLE Options > `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT`

Enable 2M-PHY

**Default value:**

- Yes (enabled) if `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT` && `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_BT_NIMBLE_LL_CFG_FEAT_LE_CODED_PHY**

Enable coded Phy

*Found in:* Component config > Bluetooth > NimBLE Options > `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT`

Enable coded-PHY

**Default value:**

- Yes (enabled) if `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT` && `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_BT_NIMBLE_EXT_ADV**

Enable extended advertising

*Found in:* Component config > Bluetooth > NimBLE Options > `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT`

Enable this option to do extended advertising. Extended advertising will be supported from BLE 5.0 onwards.

**Default value:**

- No (disabled) if `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT` && `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_BT_NIMBLE_MAX_EXT_ADV_INSTANCES**

Maximum number of extended advertising instances.

*Found in:* Component config > Bluetooth > NimBLE Options > `CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT` > `CONFIG_BT_NIMBLE_EXT_ADV`

Change this option to set maximum number of extended advertising instances. Minimum there is always one instance of advertising. Enter how many more advertising instances you want. For ESP32C2, ESP32C6 and ESP32H2, each extended advertising instance will take about 0.5k DRAM.

**Range:**

- from 0 to 4 if `CONFIG_BT_NIMBLE_EXT_ADV` && `CONFIG_BT_NIMBLE_EXT_ADV` && `CONFIG_BT_NIMBLE_ENABLED`

**Default value:**

- 1 if `CONFIG_BT_NIMBLE_EXT_ADV` && `CONFIG_BT_NIMBLE_EXT_ADV` && `CONFIG_BT_NIMBLE_ENABLED`
CONFIG_BT_NIMBLE_EXT_ADV_MAX_SIZE

Maximum length of the advertising data.

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV

Defines the length of the extended adv data. The value should not exceed 1650.

Range:
- from 0 to 1650 if CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_ENABLED

Default value:
- 1650 if CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_ENABLED
- 0 if CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV

Enable periodic advertisement.

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV

Enable this option to start periodic advertisement.

Default value:
- Yes (enabled) if CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_PERIODICADV_SYNC_TRANSFER

Enable Transfer Sync Events

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT > CONFIG_BT_NIMBLE_EXT_ADV > CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV

This enables controller transfer periodic sync events to host

Default value:
- Yes (enabled) if CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV && CONFIG_BT_NIMBLE_EXT_ADV && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_PERIODIC_SYNCS

Maximum number of periodic advertising syncs

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Set this option to set the upper limit for number of periodic sync connections. This should be less than maximum connections allowed by controller.

Range:
- from 0 to 8 if CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && CONFIG_BT_NIMBLE_ENABLED

Default value:
• 1 if CONFIG_BT_NIMBLE_ENABLE_PERIODIC_ADV && CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && CONFIG_BT_NIMBLE_ENABLED
• 0 if CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_MAX_PERIODIC_ADVERTISER_LIST

Maximum number of periodic advertiser list

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Set this option to set the upper limit for number of periodic advertiser list.

Range:
• from 1 to 5 if CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && SOC_ESP_NIMBLE_CONTROLLER && CONFIG_BT_NIMBLE_ENABLED

Default value:
• 5 if CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && SOC_ESP_NIMBLE_CONTROLLER && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_BLE_POWER_CONTROL

Enable support for BLE Power Control

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Set this option to enable the Power Control feature

Default value:
• No (disabled) if CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT && SOC_BLE_POWER_CONTROL_SUPPORTED && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_PERIODIC_ADV_ENH

Periodic adv enhancements(adi support)

Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT

Enable the periodic advertising enhancements

CONFIG_BT_NIMBLE_WHITELIST_SIZE

BLE white list size

Found in: Component config > Bluetooth > NimBLE Options

BLE list size

Range:
• from 1 to 15 if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

Default value:
• 12 if CONFIG_BT_NIMBLE_ENABLED && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_TEST_THROUGHPUT_TEST

Throughput Test Mode enable

Found in: Component config > Bluetooth > NimBLE Options

Enable the throughput test mode
Chapter 2. API Reference

Default value:
- No (disabled) if \texttt{CONFIG_BT_NIMBLE_ENABLED}

\textbf{CONFIG_BT_NIMBLE_BLUFI_ENABLE}
Enable blufi functionality

\textit{Found in: Component config > Bluetooth > NimBLE Options}

Set this option to enable blufi functionality.

Default value:
- No (disabled) if \texttt{CONFIG_BT_NIMBLE_ENABLED \&\& CONFIG_BT_NIMBLE_ENABLED}

\textbf{CONFIG_BT_NIMBLE_USE_ESP_TIMER}
Enable Esp Timer for Nimble

\textit{Found in: Component config > Bluetooth > NimBLE Options}

Set this option to use Esp Timer which has higher priority timer instead of FreeRTOS timer

Default value:
- Yes (enabled) if \texttt{CONFIG_BT_NIMBLE_ENABLED}

\textbf{CONFIG_BT_NIMBLE_BLE_GATT_BLOB_TRANSFER}
Blob transfer

\textit{Found in: Component config > Bluetooth > NimBLE Options}

This option is used when data to be sent is more than 512 bytes. For peripheral role, \texttt{BT_NIMBLE_MSYS_\_BLOCK\_COUNT} needs to be increased according to the need.

\textbf{GAP Service} Contains:
- \texttt{GAP Appearance write permissions}
- \texttt{CONFIG_BT_NIMBLE_SVC_GAP_CENT_ADDR_RESOLUTION}
- \texttt{GAP device name write permissions}
- \texttt{CONFIG_BT_NIMBLE_SVC_GAP_PPCP\_MAX\_CONN\_INTERVAL}
- \texttt{CONFIG_BT_NIMBLE_SVC_GAP_PPCP\_MIN\_CONN\_INTERVAL}
- \texttt{CONFIG_BT_NIMBLE_SVC_GAP_PPCP\_SLAVE\_LATENCY}
- \texttt{CONFIG_BT_NIMBLE_SVC_GAP_PPCP\_SUPERVISION\_TMO}

\textbf{GAP Appearance write permissions} Contains:
- \texttt{CONFIG_BT_NIMBLE_SVC_GAP\_APPEAR\_WRITE}

\textbf{CONFIG_BT_NIMBLE_SVC_GAP\_APPEAR\_WRITE}
Write

\textit{Found in: Component config > Bluetooth > NimBLE Options > GAP Service > GAP Appearance write permissions}

Enable write permission (BLE\_GATT\_CHR\_F\_WRITE)

Default value:
- No (disabled) if \texttt{CONFIG_BT_NIMBLE_ENABLED}
CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE_ENC

Write with encryption

*Found in:* Component config > Bluetooth > NimBLE Options > GAP Service > GAP Appearance write permissions > CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE

Enable write with encryption permission (BLE_GATT_CHR_F_WRITE_ENC)

**Default value:**
- No (disabled) if CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE_AUTHEN

Write with authentication

*Found in:* Component config > Bluetooth > NimBLE Options > GAP Service > GAP Appearance write permissions > CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE

Enable write with authentication permission (BLE_GATT_CHR_F_WRITE_AUTHEN)

**Default value:**
- No (disabled) if CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE_AUTHOR

Write with authorisation

*Found in:* Component config > Bluetooth > NimBLE Options > GAP Service > GAP Appearance write permissions > CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE

Enable write with authorisation permission (BLE_GATT_CHR_F_WRITE_AUTHOR)

**Default value:**
- No (disabled) if CONFIG_BT_NIMBLE_SVC_GAP_APPEAR_WRITE && CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_CENT_ADDR_RESOLUTION

GAP Characteristic - Central Address Resolution

*Found in:* Component config > Bluetooth > NimBLE Options > GAP Service

Weather or not Central Address Resolution characteristic is supported on the device, and if supported, weather or not Central Address Resolution is supported.

- Central Address Resolution characteristic not supported
- Central Address Resolution not supported
- Central Address Resolution supported

Available options:

- Characteristic not supported (CONFIG_BT_NIMBLE_SVC_GAP_CAR_CHAR_NOT_SUPP)
- Central Address Resolution not supported (CONFIG_BT_NIMBLE_SVC_GAP_CAR_NOT_SUPP)
- Central Address Resolution supported (CONFIG_BT_NIMBLE_SVC_GAP_CAR_SUPP)

GAP device name write permissions  
Contains:

- CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE
CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE

Write

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service > GAP device name write permissions*

Enable write permission (BLE_GATT_CHR_F_WRITE)

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE_ENC

Write with encryption

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service > GAP device name write permissions > CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE*

Enable write with encryption permission (BLE_GATT_CHR_F_WRITE_ENC)

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE_AUTHEN

Write with authentication

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service > GAP device name write permissions > CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE*

Enable write with authentication permission (BLE_GATT_CHR_F_WRITE_AUTHEN)

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE_AUTHOR

Write with authorisation

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service > GAP device name write permissions > CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE*

Enable write with authorisation permission (BLE_GATT_CHR_F_WRITE_AUTHOR)

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_SVC_GAP_NAME_WRITE` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_SVC_GAP_PPCP_MAX_CONN_INTERVAL

PPCP Connection Interval Max (Unit: 1.25 ms)

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service*

Peripheral Preferred Connection Parameter: Connection Interval maximum value Interval Max = value * 1.25 ms

**Default value:**
- 0 if `CONFIG_BT_NIMBLE_ROLE_PERIPHERAL` && `CONFIG_BT_NIMBLE_ENABLED`
CONFIG_BT_NIMBLE_SVC_GAP_PPCP_MIN_CONN_INTERVAL

PPCP Connection Interval Min (Unit: 1.25 ms)

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service*

Peripheral Preferred Connection Parameter: Connection Interval minimum value Interval Min = value * 1.25 ms

*Default value:*
  - 0 if CONFIG_BT_NIMBLE_ROLE_PERIPHERAL & CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_PPCP_SLAVE_LATENCY

PPCP Slave Latency

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service*

Peripheral Preferred Connection Parameter: Slave Latency

*Default value:*
  - 0 if CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_SVC_GAP_PPCP_SUPERVISION_TMO

PPCP Supervision Timeout (Uint: 10 ms)

*Found in: Component config > Bluetooth > NimBLE Options > GAP Service*

Peripheral Preferred Connection Parameter: Supervision Timeout Timeout = Value * 10 ms

*Default value:*
  - 0 if CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_VS_SUPPORT

Enable support for VSC and VSE

*Found in: Component config > Bluetooth > NimBLE Options*

This option is used to enable support for sending Vendor Specific HCI commands and handling Vendor Specific HCI Events.

CONFIG_BT_NIMBLE_OPTIMIZE_MULTI_CONN

Enable the optimization of multi-connection

*Found in: Component config > Bluetooth > NimBLE Options*

This option enables the use of vendor-specific APIs for multi-connections, which can greatly enhance the stability of coexistence between numerous central and peripheral devices. It will prohibit the usage of standard APIs.

*Default value:*
  - No (disabled) if SOC_BLE_MULTI_CONN_OPTIMIZATION & CONFIG_BT_NIMBLE_ENABLED

CONFIG_BT_NIMBLE_ENC_ADV_DATA

Encrypted Advertising Data

*Found in: Component config > Bluetooth > NimBLE Options*

This option is used to enable encrypted advertising data.
Chapter 2. API Reference

CONFIG_BT_NIMBLE_MAX_EADS

Maximum number of EAD devices to save across reboots

*Found in: Component config > Bluetooth > NimBLE Options > CONFIG_BT_NIMBLE_ENC_ADV_DATA*

Defines maximum number of encrypted advertising data key material to save

**Default value:**
- 10 if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENC_ADV_DATA` && `CONFIG_BT_NIMBLE_ENABLED`

CONFIG_BT_NIMBLE_HIGH_DUTY_ADV_ITVL

Enable BLE high duty advertising interval feature

*Found in: Component config > Bluetooth > NimBLE Options*

This enable BLE high duty advertising interval feature

CONFIG_BT_NIMBLE_HOST_QUEUE_CONG_CHECK

BLE queue congestion check

*Found in: Component config > Bluetooth > NimBLE 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.

**Default value:**
- No (disabled) if `CONFIG_BT_NIMBLE_ENABLED` && `CONFIG_BT_NIMBLE_ENABLED`

Controller Options

CONFIG_BT_RELEASE_IRAM

Release Bluetooth text (READ DOCS FIRST)

*Found in: Component config > Bluetooth*

This option release Bluetooth text section and merge Bluetooth data, bss & text into a large free heap region when esp_bt_mem_release is called, total saving ~21kB or more of IRAM. ESP32-C2 only 3 configurable PMP entries available, rest of them are hard-coded. We cannot split the memory into 3 different regions (IRAM, BLE-IRAM, DRAM). So this option will disable the PMP (ESP_SYSTEM_PMP_IDRAM_SPLIT)

**Default value:**
- No (disabled) if `CONFIG_BT_ENABLED` && `BT_LE_RELEASE_IRAM_SUPPORTED`

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:
- **BLE Mesh and BLE coexistence support**
- `CONFIG_BLE_MESH_GATT_PROXY_CLIENT`
- `CONFIG_BLE_MESH_GATT_PROXY_SERVER`
Chapter 2. API Reference

- BLE Mesh NET BUF DEBUG LOG LEVEL
- CONFIG_BLE_MESH_PROV
- CONFIG_BLE_MESH_PROXY
- BLE Mesh specific test option
- BLE Mesh STACK DEBUG LOG LEVEL
- CONFIG_BLE_MESH_NO_LOG
- CONFIG_BLE_MESH_IVU_DIVIDER
- CONFIG_BLE_MESH_FAST_PROV
- CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC
- CONFIG_BLE_MESH_CRPL
- CONFIG_BLE_MESH_RX_SDU_MAX
- CONFIG_BLE_MESH_MODEL_KEY_COUNT
- CONFIG_BLE_MESH_APP_KEY_COUNT
- CONFIG_BLE_MESH_MODEL_GROUP_COUNT
- CONFIG_BLE_MESH_LABEL_COUNT
- CONFIG_BLE_MESH_SUBNET_COUNT
- CONFIG_BLE_MESH_TX_SEG_MAX
- CONFIG_BLE_MESH_RX_SEG_MSG_COUNT
- CONFIG_BLE_MESH_MEM_ALLOC_MODE
- CONFIG_BLE_MESH_MSG_CACHE_SIZE
- CONFIG_BLE_MESH_ADV_BUF_COUNT
- CONFIG_BLE_MESH_PB_GATT
- CONFIG_BLE_MESH_PB_ADV
- CONFIG_BLE_MESH_IVU_RECOVERY_IVI
- CONFIG_BLE_MESH_RELAY
- CONFIG_BLE_MESH_SAR_ENHANCEMENT
- CONFIG_BLE_MESH_SETTINGS
- CONFIG_BLE_MESH_ACTIVE_SCAN
- CONFIG_BLE_MESH_DEINIT
- CONFIG_BLE_MESH_USE_DUPLICATE_SCAN

Support for BLE Mesh Client/Server models
Support for BLE Mesh Foundation models

- CONFIG_BLE_MESH_NODE
- CONFIG_BLE_MESH_PROVISIONER
- CONFIG_BLE_MESH_FRIEND
- CONFIG_BLE_MESH_LOW_POWER
- CONFIG_BLE_MESH_HCI_5_0
- CONFIG_BLE_MESH_IV_UPDATE_TEST
- CONFIG_BLE_MESH_CLIENT_MSG_TIMEOUT

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.

**Default value:**
- Yes (enabled) if CONFIG_BLE_MESH

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”.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_ACTIVE_SCAN**

Support Active Scan in BLE Mesh

*Found in: Component config > CONFIG_BLE_MESH*

Enable this option to allow using BLE Active Scan for BLE Mesh.

**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
- Internal IRAM memory wherever applicable else internal DRAM

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.

(*) In case of ESP32-S2/ESP32-S3, hardware allows encryption of external SPIRAM contents provided hardware flash encryption feature is enabled. In that case, using external SPIRAM allocation strategy is also safe choice from security perspective.

Available options:

- Internal DRAM (CONFIG_BLE_MESH_MEM_ALLOC_MODE_INTERNAL)
- External SPIRAM (CONFIG_BLE_MESH_MEM_ALLOC_MODE_EXTERNAL)
- Default alloc mode (CONFIG_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.
- Internal IRAM (CONFIG_BLE_MESH_MEM_ALLOC_MODE_IRAM_8BIT)
  Allows to use IRAM memory region as 8bit accessible region. Every unaligned (8bit or 16bit) access will result in an exception and incur penalty of certain clock cycles per unaligned read/write.

**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 or IRAM) 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.

**Default value:**
- No (disabled) if `ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY` && `CONFIG_BLE_MESH`
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 (CONFIG_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.

- Internal IRAM (CONFIG_BLE_MESH_FREERTOS_STATIC_ALLOC_IRAM_8BIT)
  If enabled, BLE Mesh allocates dynamic memory from internal IRAM for FreeRTOS objects, i.e. mutex, queue. Note: IRAM region cannot be used as task stack.

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.

Default value:

- Yes (enabled) if CONFIG_BLE_MESH

BLE Mesh and BLE coexistence support  Contains:

- CONFIG_BLE_MESH_SUPPORT_BLE_SCAN
- CONFIG_BLE_MESH_SUPPORT_BLE_ADV

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.

Default value:

- No (disabled) if CONFIG_BLE_MESH

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.

Range:

- from 1 to 255 if CONFIG_BLE_MESH_SUPPORT_BLE_ADV && CONFIG_BLE_MESH

Default value:

- 3 if CONFIG_BLE_MESH_SUPPORT_BLE_ADV && CONFIG_BLE_MESH
**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.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH`

**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.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH`

**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).

**Range:**
- from 1 to 100 if `CONFIG_BLE_MESH_PROVISIONER` && `CONFIG_BLE_MESH`

**Default value:**
- 10 if `CONFIG_BLE_MESH_PROVISIONER` && `CONFIG_BLE_MESH`
**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.

*Range:*
- from 1 to 1000 if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

*Default value:*
- 10 if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

**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 example, if the value is 2, it means a Provisioner can provision two unprovisioned devices with PB-ADV at the same time.

*Range:*
- from 1 to 10 if CONFIG_BLE_MESH_PB_ADV && CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

*Default value:*
- 2 if CONFIG_BLE_MESH_PB_ADV && CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

**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.

*Range:*
- from 1 to 5 if CONFIG_BLE_MESH_PB_GATT && CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

*Default value:*
- 1 if CONFIG_BLE_MESH_PB_GATT && CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

**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.

*Range:*
- from 1 to 4096 if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

*Default value:*

Chapter 2. API Reference

• 3 if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

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.

Range:
• from 1 to 4096 if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

Default value:
• 3 if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

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.

Default value:
• No (disabled) if CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

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.

Range:
• from 1 to 1000 if CONFIG_BLE_MESH_PROVISIONER_RECV_HB && CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

Default value:
• 3 if CONFIG_BLE_MESH_PROVISIONER_RECV_HB && CONFIG_BLE_MESH_PROVISIONER && CONFIG_BLE_MESH

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.

Default value:
• Yes (enabled) if CONFIG_BLE_MESH
**CONFIG_BLE_MESH_PROV_EPA**

BLE Mesh enhanced provisioning authentication

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROV*

Enable this option to support BLE Mesh enhanced provisioning authentication functionality. This option can increase the security level of provisioning. It is recommended to enable this option.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH_PROV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_CERT_BASED_PROV**

Support Certificate-based provisioning

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROV*

Enable this option to support BLE Mesh Certificate-Based Provisioning.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH_PROV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_RECORD_FRAG_MAX_SIZE**

Maximum size of the provisioning record fragment that Provisioner can receive

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_PROV > CONFIG_BLE_MESH_CERT_BASED_PROV*

This option sets the maximum size of the provisioning record fragment that the Provisioner can receive. The range depends on provisioning bearer.

**Range:**
- from 1 to 57 if `CONFIG_BLE_MESH_CERT_BASED_PROV` && `CONFIG_BLE_MESH`

**Default value:**
- 56 if `CONFIG_BLE_MESH_CERT_BASED_PROV` && `CONFIG_BLE_MESH`

**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.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH`

**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.

**Range:**
- from 1 to 100 if `CONFIG_BLE_MESH_NODE` && `CONFIG_BLE_MESH_PB_ADV` && `CONFIG_BLE_MESH`

**Default value:**
Chapter 2. API Reference

• 5 if `CONFIG_BLE_MESH_NODE` && `CONFIG_BLE_MESH_PB_ADV` && `CONFIG_BLE_MESH`

• 3 if `CONFIG_BLE_MESH_FAST_PROV` && `CONFIG_BLE_MESH_NODE` && `CONFIG_BLE_MESH_PB_ADV` && `CONFIG_BLE_MESH`

**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.

**Default value:**

• Yes (enabled) if `CONFIG_BLE_MESH`

**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.

**Default value:**

• Yes (enabled) if `CONFIG_BLE_MESH_NODE` && `CONFIG_BLE_MESH`

**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.

**Range:**

• from 1 to 60 if `CONFIG_BLE_MESH_GATT_PROXY_SERVER` && `CONFIG_BLE_MESH`

**Default value:**

• 60 if `CONFIG_BLE_MESH_GATT_PROXY_SERVER` && `CONFIG_BLE_MESH`
**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.

**Range:**

- from 1 to 32767 if CONFIG_BLE_MESH_GATT_PROXY_SERVER && CONFIG_BLE_MESH

**Default value:**

- 4 if CONFIG_BLE_MESH_GATT_PROXY_SERVER && CONFIG_BLE_MESH

**CONFIG_BLE_MESH_PROXY_PRIVACY**

Support Proxy Privacy

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_SERVER*

The Proxy Privacy parameter controls the privacy of the Proxy Server over the connection. The value of the Proxy Privacy parameter is controlled by the type of proxy connection, which is dependent on the bearer used by the proxy connection.

**Default value:**

- Yes (enabled) if CONFIG_BLE_MESH_PRB_SRV && CONFIG_BLE_MESH_GATT_PROXY_SERVER && CONFIG_BLE_MESH

**CONFIG_BLE_MESH_PROXY_SOLIC_PDU_RX**

Support receiving Proxy Solicitation PDU

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_SERVER*

Enable this option to support receiving Proxy Solicitation PDU.

**CONFIG_BLE_MESH_PROXY_SOLIC_RX_CRPL**

Maximum capacity of solicitation replay protection list

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_SERVER > CONFIG_BLE_MESH_PROXY_SOLIC_PDU_RX*

This option specifies the maximum capacity of the solicitation replay protection list. The solicitation replay protection list is used to reject Solicitation PDUs that were already processed by a node, which will store the solicitation src and solicitation sequence number of the received Solicitation PDU message.

**Range:**

- from 1 to 255 if CONFIG_BLE_MESH_PROXY_SOLIC_PDU_RX && CONFIG_BLE_MESH

**Default value:**

- 2 if CONFIG_BLE_MESH_PROXY_SOLIC_PDU_RX && CONFIG_BLE_MESH

**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.

**Default value:**

- No (disabled) if CONFIG_BLE_MESH
CONFIG_BLE_MESH_PROXY_SOLIC_PDU_TX

Support sending Proxy Solicitation PDU

*Found in:* Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_CLIENT

Enable this option to support sending Proxy Solicitation PDU.

CONFIG_BLE_MESH_PROXY_SOLIC_TX_SRC_COUNT

Maximum number of SSRC that can be used by Proxy Client

*Found in:* Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_GATT_PROXY_CLIENT
> CONFIG_BLE_MESH_PROXY_SOLIC_PDU_TX

This option specifies the maximum number of Solicitation Source (SSRC) that can be used by Proxy Client for sending a Solicitation PDU. A Proxy Client may use the primary address or any of the secondary addresses as the SSRC for a Solicitation PDU. So for a Proxy Client, it’s better to choose the value based on its own element count.

**Range:**
- from 1 to 16 if CONFIG_BLE_MESH_PROXY_SOLIC_PDU_TX && CONFIG_BLE_MESH

**Default value:**
- 2 if CONFIG_BLE_MESH_PROXY_SOLIC_PDU_TX && CONFIG_BLE_MESH

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.

**Default value:**
- No (disabled) if CONFIG_BLE_MESH

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.

**Range:**
- from 0 to 1000000 if CONFIG_BLE_MESH_SETTINGS && CONFIG_BLE_MESH

**Default value:**
- 0 if CONFIG_BLE_MESH_SETTINGS && CONFIG_BLE_MESH

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.

**Range:**
- from 0 to 1000000 if `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**Default value:**
- 0 if `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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.

**Range:**
- from 0 to 1000000 if `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**Default value:**
- 0 if `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH_NODE` && `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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 exists 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.
**Default value:**
- No (disabled) if `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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.

**Default value:**
- “ble_mesh” if `CONFIG_BLE_MESH_SPECIFIC_PARTITION` && `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH_PROVISIONER` && `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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.

**Range:**
- from 1 to 255 if `CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE` && `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**Default value:**
- 2 if `CONFIG_BLE_MESH_USE_MULTIPLE_NAMESPACE` && `CONFIG_BLE_MESH_SETTINGS` && `CONFIG_BLE_MESH`

**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.

**Range:**
- from 1 to 4096 if `CONFIG_BLE_MESH`

**Default value:**
• 3 if CONFIG_BLE_MESH

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.

*Range:*
  • from 1 to 4096 if CONFIG_BLE_MESH

*Default value:*
  • 3 if CONFIG_BLE_MESH

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.

*Range:*
  • from 1 to 4096 if CONFIG_BLE_MESH

*Default value:*
  • 3 if CONFIG_BLE_MESH

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.

*Range:*
  • from 1 to 4096 if CONFIG_BLE_MESH

*Default value:*
  • 3 if CONFIG_BLE_MESH

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 that can be supported by a node.

*Range:*
  • from 0 to 4096 if CONFIG_BLE_MESH

*Default value:*
  • 3 if CONFIG_BLE_MESH

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 replaying messages.
Chapter 2. API Reference

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.

**Range:**
- from 2 to 65535 if `CONFIG_BLE_MESH`

**Default value:**
- 10 if `CONFIG_BLE_MESH`

**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 opera-
tions 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.

**Range:**
- from 2 to 65535 if `CONFIG_BLE_MESH`

**Default value:**
- 10 if `CONFIG_BLE_MESH`

**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.

**Range:**
- from 6 to 256 if `CONFIG_BLE_MESH`

**Default value:**
- 60 if `CONFIG_BLE_MESH`

**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).
Range:
- from 2 to 96 if CONFIG_BLE_MESH

Default value:
- 4 if CONFIG_BLE_MESH

**CONFIG_BLE_MESH_IVU_RECOVERY_IVI**

Recovery the IV index when the latest whole IV update procedure is missed

*Found in: Component config > CONFIG_BLE_MESH*

According to Section 3.10.5 of Mesh Specification v1.0.1. If a node in Normal Operation receives a Secure Network beacon with an IV index equal to the last known IV index+1 and the IV Update Flag set to 0, the node may update its IV without going to the IV Update in Progress state, or it may initiate an IV Index Recovery procedure (Section 3.10.6), or it may ignore the Secure Network beacon. The node makes the choice depending on the time since last IV update and the likelihood that the node has missed the Secure Network beacons with the IV update Flag. When the above situation is encountered, this option can be used to decide whether to perform the IV index recovery procedure.

Default value:
- No (disabled) if CONFIG_BLE_MESH

**CONFIG_BLE_MESH_SAR_ENHANCEMENT**

Segmentation and reassembly enhancement

*Found in: Component config > CONFIG_BLE_MESH*

Enable this option to use the enhanced segmentation and reassembly mechanism introduced in Bluetooth Mesh Protocol 1.1.

Default value:
- No (disabled) if CONFIG_BLE_MESH

**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.

Range:
- from 1 to if CONFIG_BLE_MESH

Default value:
- 1 if CONFIG_BLE_MESH

**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.

Range:
**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.

*Range:*

- from 36 to 384 if CONFIG_BLE_MESH

*Default value:*

- 384 if CONFIG_BLE_MESH

**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.

*Range:*

- from 2 to 32 if CONFIG_BLE_MESH

*Default value:*

- 32 if CONFIG_BLE_MESH

**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.

*Default value:*

- Yes (enabled) if CONFIG_BLE_MESH_NODE && CONFIG_BLE_MESH

**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.
Default value:  
- No (disabled) if `CONFIG_BLE_MESH_RELAY` && `CONFIG_BLE_MESH`

**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.  
**Range:**  
- from 6 to 256 if `CONFIG_BLE_MESH_RELAY_ADV_BUF` && `CONFIG_BLE_MESH_RELAY` && `CONFIG_BLE_MESH`  
**Default value:**  
- 60 if `CONFIG_BLE_MESH_RELAY_ADV_BUF` && `CONFIG_BLE_MESH_RELAY` && `CONFIG_BLE_MESH`

**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.  
**Default value:**  
- No (disabled) if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`

**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 to send Friend Request at a certain period until it finds a proper Friend node.  
**Default value:**  
- No (disabled) if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`

**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.

**Range:**
- from 0 to 3600 if `CONFIG_BLE_MESH_LPN_AUTO` && `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**Default value:**
- 15 if `CONFIG_BLE_MESH_LPN_AUTO` && `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**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.

**Range:**
- from 1 to 3600 if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**Default value:**
- 6 if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**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.

**Range:**
- from 0 to 3 if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**Default value:**
- 0 if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**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.

**Range:**
- from 0 to 3 if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**Default value:**
- 0 if `CONFIG_BLE_MESH_LOW_POWER` && `CONFIG_BLE_MESH`  

**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.

Range:
- from 10 to 255 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

Default value:
- 100 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

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.

Range:
- from 10 to 244735 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

Default value:
- 300 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

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.

Range:
- from 10 to if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

Default value:
- if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

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.

Range:
- from 0 to 50 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

Default value:
• 10 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

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.

*Range:*
  • from 0 to 16384 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

*Default value:*
  • 8 if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

CONFIG_BLE_MESH_LPN_SUB_ALL_NODES_ADDR
Automatically subscribe all nodes address

*Found in: Component config > CONFIG_BLE_MESH > CONFIG_BLE_MESH_LOW_POWER*

Automatically subscribe all nodes address when friendship established.

*Default value:*
  • No (disabled) if CONFIG_BLE_MESH_LOW_POWER && CONFIG_BLE_MESH

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.

*Range:*
  • from 1 to 255 if CONFIG_BLE_MESH_FRIEND && CONFIG_BLE_MESH

*Default value:*
  • 255 if CONFIG_BLE_MESH_FRIEND && CONFIG_BLE_MESH

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.

*Range:*
  • from 2 to 65536 if CONFIG_BLE_MESH_FRIEND && CONFIG_BLE_MESH

*Default value:*
  • 16 if CONFIG_BLE_MESH_FRIEND && CONFIG_BLE_MESH
**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.

*Range:*
  - from 0 to 1023 if `CONFIG_BLE_MESH_FRIEND` && `CONFIG_BLE_MESH`

*Default value:*
  - 3 if `CONFIG_BLE_MESH_FRIEND` && `CONFIG_BLE_MESH`

**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.

*Range:*
  - from 1 to 1000 if `CONFIG_BLE_MESH_FRIEND` && `CONFIG_BLE_MESH`

*Default value:*
  - 2 if `CONFIG_BLE_MESH_FRIEND` && `CONFIG_BLE_MESH`

**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 Friend’s LPN. In other words, this determines from how many elements can segmented messages destined for the Friend queue be received simultaneously.

*Range:*
  - from 1 to 1000 if `CONFIG_BLE_MESH_FRIEND` && `CONFIG_BLE_MESH`

*Default value:*
  - 1 if `CONFIG_BLE_MESH_FRIEND` && `CONFIG_BLE_MESH`

**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.

*Default value:*
  - No (disabled) if `CONFIG_BLE_MESH` && `CONFIG_BLE_MESH`

**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 (CONFIG_BLE_MESH_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BLE_MESH_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BLE_MESH_TRACE_LEVEL_WARNING)
- INFO (CONFIG_BLE_MESH_TRACE_LEVEL_INFO)
- DEBUG (CONFIG_BLE_MESH_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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 (CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL_NONE)
- ERROR (CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL_ERROR)
- WARNING (CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL_WARNING)
- INFO (CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL_INFO)
- DEBUG (CONFIG_BLE_MESH_NET_BUF_TRACE_LEVEL_DEBUG)
- VERBOSE (CONFIG_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.

*Range:*

- from 100 to 1200000 if CONFIG_BLE_MESH

*Default value:*

- 4000 if CONFIG_BLE_MESH

Support for BLE Mesh Foundation models

*Contains:*

- CONFIG_BLE_MESH_BRC_CLI
- CONFIG_BLE_MESH_BRC_SRV
- CONFIG_BLE_MESH_CFG_CLI
- CONFIG_BLE_MESH_DF_CLI
• `CONFIG_BLE_MESH_DF_SRV`
• `CONFIG_BLE_MESH_HEALTH_CLI`
• `CONFIG_BLE_MESH_HEALTH_SRV`
• `CONFIG_BLE_MESH_LCD_CLI`
• `CONFIG_BLE_MESH_LCD_SRV`
• `CONFIG_BLE_MESH_PRB_CLI`
• `CONFIG_BLE_MESH_PRB_SRV`
• `CONFIG_BLE_MESH_ODP_CLI`
• `CONFIG_BLE_MESH_ODP_SRV`
• `CONFIG_BLE_MESH_AGG_CLI`
• `CONFIG_BLE_MESH_AGG_SRV`
• `CONFIG_BLE_MESH_RPR_CLI`
• `CONFIG_BLE_MESH_RPR_SRV`
• `CONFIG_BLE_MESH_SAR_CLI`
• `CONFIG_BLE_MESH_SAR_SRV`
• `CONFIG_BLE_MESH_SRPL_CLI`
• `CONFIG_BLE_MESH_SRPL_SRV`
• `CONFIG_BLE_MESH_COMP_DATA_1`
• `CONFIG_BLE_MESH_COMP_DATA_128`
• `CONFIG_BLE_MESH_MODELS_METADATA_0`

`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.

Default value:

• Yes (enabled) if `CONFIG_BLE_MESH`

`CONFIG_BLE_MESH_BRC_CLI`

Bridge Configuration Client model

Found in: `Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models`

Enable support for Bridge Configuration Client model.

`CONFIG_BLE_MESH_BRC_SRV`

Bridge Configuration Server model

Found in: `Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models`

Enable support for Bridge Configuration Server model.
**Default value:**
- No (disabled) if `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_MAX_BRIDGING_TABLE_ENTRY_COUNT**

Maximum number of Bridging Table entries

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_BRC_SRV*

Maximum number of Bridging Table entries that the Bridge Configuration Server can support.

**Range:**
- from 16 to 65535 if `CONFIG_BLE_MESH_BRC_SRV` && `CONFIG_BLE_MESH`

**Default value:**
- 16 if `CONFIG_BLE_MESH_BRC_SRV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_BRIDGE_CRPL**

Maximum capacity of bridge replay protection list

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_BRC_SRV*

This option specifies the maximum capacity of the bridge replay protection list. The bridge replay protection list is used to prevent a bridged subnet from replay attack, which will store the source address and sequence number of the received bridge messages.

**Range:**
- from 1 to 255 if `CONFIG_BLE_MESH_BRC_SRV` && `CONFIG_BLE_MESH`

**Default value:**
- 5 if `CONFIG_BLE_MESH_BRC_SRV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_PRB_CLI**

Mesh Private Beacon Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Mesh Private Beacon Client model.

**CONFIG_BLE_MESH_PRB_SRV**

Mesh Private Beacon Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Mesh Private Beacon Server model.

**CONFIG_BLE_MESH_ODP_CLI**

On-Demand Private Proxy Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for On-Demand Private Proxy Client model.

**CONFIG_BLE_MESH_ODP_SRV**

On-Demand Private Proxy Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for On-Demand Private Proxy Server model.
## CONFIG_BLE_MESH_SRPL_CLI
Solicitation PDU RPL Configuration Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Solicitation PDU RPL Configuration Client model.

## CONFIG_BLE_MESH_SRPL_SRV
Solicitation PDU RPL Configuration Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Solicitation PDU RPL Configuration Server model. Note: This option depends on the functionality of receiving Solicitation PDU. If the device doesn’t support receiving Solicitation PDU, then there is no need to enable this server model.

## CONFIG_BLE_MESH_AGG_CLI
Opcodes Aggregator Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Opcodes Aggregator Client model.

## CONFIG_BLE_MESH_AGG_SRV
Opcodes Aggregator Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Opcodes Aggregator Server model.

## CONFIG_BLE_MESH_SAR_CLI
SAR Configuration Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for SAR Configuration Client model.

## CONFIG_BLE_MESH_SAR_SRV
SAR Configuration Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for SAR Configuration Server model.

## CONFIG_BLE_MESH_COMP_DATA_1
Support Composition Data Page 1

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Composition Data Page 1 contains information about the relationships among models. Each model either can be a root model or can extend other models.
CONFIG_BLE_MESH_COMP_DATA_128

Support Composition Data Page 128

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Composition Data Page 128 is used to indicate the structure of elements, features, and models of a node after the successful execution of the Node Address Refresh procedure or the Node Composition Refresh procedure, or after the execution of the Node Removal procedure followed by the provisioning process. Composition Data Page 128 shall be present if the node supports the Remote Provisioning Server model; otherwise it is optional.

CONFIG_BLE_MESH_MODELS_METADATA_0

Support Models Metadata Page 0

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

The Models Metadata state contains metadata of a node’s models. The Models Metadata state is composed of a number of pages of information. Models Metadata Page 0 shall be present if the node supports the Large Composition Data Server model.

CONFIG_BLE_MESH_MODELS_METADATA_128

Support Models Metadata Page 128

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_MODELS_METADATA_0

The Models Metadata state contains metadata of a node’s models. The Models Metadata state is composed of a number of pages of information. Models Metadata Page 128 contains metadata for the node’s models after the successful execution of the Node Address Refresh procedure or the Node Composition Refresh procedure, or after the execution of the Node Removal procedure followed by the provisioning process. Models Metadata Page 128 shall be present if the node supports the Remote Provisioning Server model and the node supports the Large Composition Data Server model.

CONFIG_BLE_MESH_LCD_CLI

Large Composition Data Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Large Composition Data Client model.

CONFIG_BLE_MESH_LCD_SRV

Large Composition Data Server model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Large Composition Data Server model.

CONFIG_BLE_MESH_RPR_CLI

Remote Provisioning Client model

Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Remote Provisioning Client model
**CONFIG_BLE_MESH_RPR_CLI_PROV_SAME_TIME**

Maximum number of PB-Remote running at the same time by Provisioner

*Found in:* Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_RPR_CLI

This option specifies how many devices can be provisioned at the same time using PB-REMOTE. For example, if the value is 2, it means a Provisioner can provision two unprovisioned devices with PB-REMOTE at the same time.

*Range:*
  - from 1 to 5 if CONFIG_BLE_MESH_RPR_CLI && CONFIG_BLE_MESH

*Default value:*
  - 2 if CONFIG_BLE_MESH_RPR_CLI && CONFIG_BLE_MESH

**CONFIG_BLE_MESH_RPR_SRV**

Remote Provisioning Server model

*Found in:* Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models

Enable support for Remote Provisioning Server model

**CONFIG_BLE_MESH_RPR_SRV_MAX_SCANNED_ITEMS**

Maximum number of device information can be scanned

*Found in:* Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_RPR_SRV

This option specifies how many device information can a Remote Provisioning Server store each time while scanning.

*Range:*
  - from 4 to 255 if CONFIG_BLE_MESH_RPR_SRV && CONFIG_BLE_MESH

*Default value:*
  - 10 if CONFIG_BLE_MESH_RPR_SRV && CONFIG_BLE_MESH

**CONFIG_BLE_MESH_RPR_SRV_ACTIVE_SCAN**

Support Active Scan for remote provisioning

*Found in:* Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_RPR_SRV

Enable this option to support Active Scan for remote provisioning.

**CONFIG_BLE_MESH_RPR_SRV_MAX_EXT_SCAN**

Maximum number of extended scan procedures

*Found in:* Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_RPR_SRV

This option specifies how many extended scan procedures can be started by the Remote Provisioning Server.

*Range:*
  - from 1 to 10 if CONFIG_BLE_MESH_RPR_SRV && CONFIG_BLE_MESH

*Default value:*
  - 1 if CONFIG_BLE_MESH_RPR_SRV && CONFIG_BLE_MESH
**CONFIG_BLE_MESH_DF_CLI**

Directed Forwarding Configuration Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Directed Forwarding Configuration Client model.

**CONFIG_BLE_MESH_DF_SRV**

Directed Forwarding Configuration Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models*

Enable support for Directed Forwarding Configuration Server model.

**CONFIG_BLE_MESH_MAX_DISC_TABLE_ENTRY_COUNT**

Maximum number of discovery table entries in a given subnet

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_DF_SRV*

Maximum number of Discovery Table entries supported by the node in a given subnet.

**Range:**
- from 2 to 255 if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**Default value:**
- 2 if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_MAX_FORWARD_TABLE_ENTRY_COUNT**

Maximum number of forward table entries in a given subnet

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_DF_SRV*

Maximum number of Forward Table entries supported by the node in a given subnet.

**Range:**
- from 2 to 64 if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**Default value:**
- 2 if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_MAX_DEPS_NODES_PER_PATH**

Maximum number of dependent nodes per path

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_DF_SRV*

Maximum size of dependent nodes list supported by each forward table entry.

**Range:**
- from 2 to 64 if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**Default value:**
- 2 if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_PATH_MONITOR_TEST**

Enable Path Monitoring test mode

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_DF_SRV*
The option only removes the Path Use timer; all other behavior of the device is not changed. If Path Monitoring test mode is going to be used, this option should be enabled.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_SUPPORT_DIRECTED_PROXY**

Enable Directed Proxy functionality

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Foundation models > CONFIG_BLE_MESH_DF_SRV*

Support Directed Proxy functionality.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH_GATT_PROXY_SERVER` && `CONFIG_BLE_MESH_DF_SRV` && `CONFIG_BLE_MESH`

**Support for BLE Mesh Client/Server models**

Contains:
- `CONFIG_BLE_MESH_MBT_CLI`
- `CONFIG_BLE_MESH_MBT_SRV`
- `CONFIG_BLE_MESH_GENERIC_BATTERY_CLI`
- `CONFIG_BLE_MESH_GENERIC_DEF_TRANS_TIME_CLI`
- `CONFIG_BLE_MESH_GENERIC_LEVEL_CLI`
- `CONFIG_BLE_MESH_GENERIC_LOCATION_CLI`
- `CONFIG_BLE_MESH_GENERIC_ONOFF_CLI`
- `CONFIG_BLE_MESH_GENERIC_POWER_LEVEL_CLI`
- `CONFIG_BLE_MESH_GENERIC_POWER_ONOFF_CLI`
- `CONFIG_BLE_MESH_GENERIC_PROPERTY_CLI`
- `CONFIG_BLE_MESH_GENERIC_SERVER`
- `CONFIG_BLE_MESH_LIGHT_CTL_CLI`
- `CONFIG_BLE_MESH_LIGHT_HSL_CLI`
- `CONFIG_BLE_MESH_LIGHT_LC_CLI`
- `CONFIG_BLE_MESH_LIGHT_LIGHTNESS_CLI`
- `CONFIG_BLE_MESH_LIGHT_XYL_CLI`
- `CONFIG_BLE_MESH_LIGHTING_SERVER`
- `CONFIG_BLE_MESH_SCENE_CLI`
- `CONFIG_BLE_MESH_SCHEDULER_CLI`
- `CONFIG_BLE_MESH_SENSOR_CLI`
- `CONFIG_BLE_MESH_SENSOR_SERVER`
- `CONFIG_BLE_MESH_TIME_SCENE_SERVER`
- `CONFIG_BLE_MESH_TIME_CLI`

**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.

**Default value:**

- Yes (enabled) if CONFIG_BLE_MESH
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.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH`

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.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH`

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.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH`

CONFIG_BLE_MESH_MBT_CLI
BLOB Transfer Client model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models*

Enable support for BLOB Transfer Client model.

**Default value:**
- Yes (enabled) if `CONFIG_BLE_MESH`

CONFIG_BLE_MESH_MAX_BLOB_RECEIVERS
Maximum number of simultaneous blob receivers

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models > CONFIG_BLE_MESH_MBT_CLI*

Maximum number of BLOB Transfer Server models that can participating in the BLOB transfer with a BLOB Transfer Client model.

**Range:**
- from 1 to 255 if `CONFIG_BLE_MESH_MBT_CLI` && `CONFIG_BLE_MESH`

**Default value:**
- 2 if `CONFIG_BLE_MESH_MBT_CLI` && `CONFIG_BLE_MESH`

CONFIG_BLE_MESH_MBT_SRV
BLOB Transfer Server model

*Found in: Component config > CONFIG_BLE_MESH > Support for BLE Mesh Client/Server models*

Enable support for BLOB Transfer Server model.
Default value:
• Yes (enabled) if `CONFIG_BLE_MESH`

**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.

Default value:
• No (disabled) if `CONFIG_BLE_MESH`

**BLE Mesh specific test option** Contains:

• `CONFIG_BLE_MESH_DEBUG`
• `CONFIG_BLE_MESH_SHELL`
• `CONFIG_BLE_MESH_BQB_TEST`
• `CONFIG_BLE_MESH_SELF_TEST`
• `CONFIG_BLE_MESH_TEST_AUTO_ENTER_NETWORK`
• `CONFIG_BLE_MESH_TEST_USE_WHITE_LIST`

**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.

Default value:
• No (disabled) if `CONFIG_BLE_MESH`

**CONFIG_BLE_MESH_BQB_TEST**

Enable BLE Mesh specific internal test

*Found in: Component config > **CONFIG_BLE_MESH > BLE Mesh specific test option***

This option is used to enable some internal functions for auto-pts test.

Default value:
• No (disabled) if `CONFIG_BLE_MESH`

**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***

With this option enabled, an unprovisioned device can automatically enters mesh network using a specific test function without the pro- visioning procedure. And on the Provisioner side, a test function needs to be invoked to add the node information into the mesh stack.

Default value:
• Yes (enabled) if `CONFIG_BLE_MESH_SELF_TEST` & `CONFIG_BLE_MESH`
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*

With this option enabled, users can use white list to filter mesh advertising packets while scanning.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH_SELF_TEST` & `CONFIG_BLE_MESH`

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.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH`

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.

**Default value:**
- No (disabled) if `CONFIG_BLE_MESH`

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.

Driver Configurations

Contains:

- Analog Comparator Configuration
- DAC Configuration
- I2C Configuration
- I2S Configuration
- LEDC Configuration
- Legacy ADC Configuration
- Parallel IO Configuration
- RMT Configuration
- Sigma Delta Modulator Configuration
- Temperature sensor Configuration
- TWAI Configuration
- UART Configuration
- USB Serial/JTAG Configuration

Legacy ADC Configuration

Contains:

- CONFIG_ADC_DISABLE_DAC
- Legacy ADC Calibration Configuration
- CONFIG_ADC_SUPPRESS_DEPRECATE_WARN

CONFIG_ADC_DISABLE_DAC

Disable DAC when ADC2 is used on GPIO 25 and 26

*Found in:* Component config > Driver Configurations > Legacy 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.

**Default value:**

- Yes (enabled)

CONFIG_ADC_SUPPRESS_DEPRECATE_WARN

Suppress legacy driver deprecated warning

*Found in:* Component config > Driver Configurations > Legacy ADC Configuration

Whether to suppress the deprecation warnings when using legacy adc driver (driver/adc.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**

- No (disabled)

Legacy ADC Calibration Configuration

Contains:

- CONFIG_ADC_CALI_SUPPRESS_DEPRECATE_WARN
**CONFIG_ADC_CALI_SUPPRESS_DEPRECATED_WARN**

Suppress legacy driver deprecated warning

*Found in: Component config > Driver Configurations > Legacy ADC Configuration > Legacy ADC Calibration Configuration*

Whether to suppress the deprecation warnings when using legacy adc calibration driver (esp_adc_cal.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**
- No (disabled)

**TWAI Configuration** Contains:
- `CONFIG_TWAI_ERRATA_FIX_LISTEN_ONLY_DOM`
- `CONFIG_TWAI_ISR_IN_IRAM`

**CONFIG_TWAI_ISR_IN_IRAM**

Place TWAI ISR function into IRAM

*Found in: Component config > Driver Configurations > TWAI Configuration*

Place the TWAI 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 TWAI_ALERT_AND_LOG flag) will have no effect.

**Default value:**
- No (disabled)

**CONFIG_TWAI_ERRATA_FIX_LISTEN_ONLY_DOM**

Add SW workaround for listen only transmits dominant bit errata

*Found in: Component config > Driver Configurations > TWAI Configuration*

When in the listen only mode, the TWAI controller must not influence the TWAI bus (i.e., must not send any dominant bits). However, while in listen only mode on the ESP32/ESP32-S2/ESP32-S3/ESP32-C3, the TWAI controller will still transmit dominant bits when it detects an error (i.e., as part of an active error frame). Enabling this option will add a workaround that forces the TWAI controller into an error passive state on initialization, thus preventing any dominant bits from being sent.

**Default value:**
- Yes (enabled)

**Temperature sensor Configuration** Contains:
- `CONFIG_TEMP_SENSOR_ENABLE_DEBUG_LOG`
- `CONFIG_TEMP_SENSOR_SUPPRESS_DEPRECATED_WARN`
- `CONFIG_TEMP_SENSOR_ISR_IRAM_SAFE`

**CONFIG_TEMP_SENSOR_SUPPRESS_DEPRECATED_WARN**

Suppress legacy driver deprecated warning

*Found in: Component config > Driver Configurations > Temperature sensor Configuration*
Wether to suppress the deprecation warnings when using legacy temperature sensor driver (driver/temp_sensor.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**
- No (disabled)

**CONFIG_TEMP_SENSOR_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > Driver Configurations > Temperature sensor Configuration*

Wether to enable the debug log message for temperature sensor driver. Note that, this option only controls the temperature sensor driver log, won’t affect other drivers.

**Default value:**
- No (disabled)

**CONFIG_TEMP_SENSOR_ISR_IRAM_SAFE**

Temperature sensor ISR IRAM-Safe

*Found in: Component config > Driver Configurations > Temperature sensor Configuration*

Ensure the Temperature Sensor interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**
- No (disabled) if SOC_TEMPERATURE_SENSOR_INTR_SUPPORT

**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.

**Sigma Delta Modulator Configuration** Contains:

*CONFIG_SD_M_ENABLE_DEBUG_LOG*
*CONFIG_SD_M_CTRL_FUNC_IN_IRAM*
*CONFIG_SD_M_SUPPRESS_DEPRECATE_WARN*

**CONFIG_SD_M_CTRL_FUNC_IN_IRAM**

Place SDM control functions into IRAM

*Found in: Component config > Driver Configurations > Sigma Delta Modulator Configuration*

Place SDM control functions (like set_duty) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

**Default value:**
- No (disabled)
CONFIG_SDM_SUPPRESS_DEPRECATED_WARN
Suppress legacy driver deprecated warning

*Found in: Component config > Driver Configurations > Sigma Delta Modulator Configuration*

Wether to suppress the deprecation warnings when using legacy sigma delta driver. If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**
- No (disabled)

CONFIG_SDM_ENABLE_DEBUG_LOG
Enable debug log

*Found in: Component config > Driver Configurations > Sigma Delta Modulator Configuration*

Wether to enable the debug log message for SDM driver. Note that, this option only controls the SDM driver log, won’t affect other drivers.

**Default value:**
- No (disabled)

Analog Comparator Configuration  Contains:
- `CONFIG_ANA_CMPR_ISR_IRAM_SAFE`
- `CONFIG_ANA_CMPR_ENABLE_DEBUG_LOG`
- `CONFIG_ANA_CMPR_CTRL_FUNC_IN_IRAM`

CONFIG_ANA_CMPR_ISR_IRAM_SAFE
Analog comparator ISR IRAM-Safe

*Found in: Component config > Driver Configurations > Analog Comparator Configuration*

Ensure the Analog Comparator interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**
- No (disabled) if SOC_ANA_CMPR_SUPPORTED

CONFIG_ANA_CMPR_CTRL_FUNC_IN_IRAM
Place Analog Comparator control functions into IRAM

*Found in: Component config > Driver Configurations > Analog Comparator Configuration*

Place Analog Comparator control functions (like ana_cmpr_set_internal_reference) into IRAM, so that these functions can be IRAM-safe and able to be called in an IRAM interrupt context. Enabling this option can improve driver performance as well.

**Default value:**
- No (disabled) if SOC_ANA_CMPR_SUPPORTED

CONFIG_ANA_CMPR_ENABLE_DEBUG_LOG
Enable debug log

*Found in: Component config > Driver Configurations > Analog Comparator Configuration*

Wether to enable the debug log message for Analog Comparator driver. Note that, this option only controls the Analog Comparator driver log, won’t affect other drivers.
Default value:
• No (disabled) if SOC_ANA_CMPR_SUPPORTED

RMT Configuration Contains:
• CONFIG_RMT_ENABLE_DEBUG_LOG
• CONFIG_RMT_RECV_FUNC_IN_IRAM
• CONFIG_RMT_ISR_IRAM_SAFE
• CONFIG_RMT_SUPPRESS_DEPRECATE_WARN

CONFIG_RMT_ISR_IRAM_SAFE
RMT ISR IRAM-Safe

*Found in: Component config > Driver Configurations > RMT Configuration*

Ensure the RMT interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

Default value:
• No (disabled)

CONFIG_RMT_RECV_FUNC_IN_IRAM
Place RMT receive function into IRAM

*Found in: Component config > Driver Configurations > RMT Configuration*

Place RMT receive function into IRAM, so that the receive function can be IRAM-safe and able to be called when the flash cache is disabled. Enabling this option can improve driver performance as well.

Default value:
• No (disabled)

CONFIG_RMT_SUPPRESS_DEPRECATE_WARN
Suppress legacy driver deprecated warning

*Found in: Component config > Driver Configurations > RMT Configuration*

Whether to suppress the deprecation warnings when using legacy rmt driver (driver/rmt.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

Default value:
• No (disabled)

CONFIG_RMT_ENABLE_DEBUG_LOG
Enable debug log

*Found in: Component config > Driver Configurations > RMT Configuration*

Whether to enable the debug log message for RMT driver. Note that, this option only controls the RMT driver log, won’t affect other drivers.

Default value:
• No (disabled)
Chapter 2. API Reference

**I2S Configuration**  Contains:

- `CONFIG_I2S_ENABLE_DEBUG_LOG`
- `CONFIG_I2S_ISR_IRAM_SAFE`
- `CONFIG_I2S_SUPPRESS_DEPRECIATE_WARN`

**CONFIG_I2S_ISR_IRAM_SAFE**

I2S ISR IRAM-Safe

*Found in: Component config > Driver Configurations > I2S Configuration*

Ensure the I2S interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**
- No (disabled)

**CONFIG_I2S_SUPPRESS_DEPRECIATE_WARN**

Suppress legacy driver deprecated warning

*Found in: Component config > Driver Configurations > I2S Configuration*

Enable this option will suppress the deprecation warnings of using APIs in legacy I2S driver.

**Default value:**
- No (disabled)

**CONFIG_I2S_ENABLE_DEBUG_LOG**

Enable I2S debug log

*Found in: Component config > Driver Configurations > I2S Configuration*

Wether to enable the debug log message for I2S driver. Note that, this option only controls the I2S driver log, will not affect other drivers.

**Default value:**
- No (disabled)

**DAC Configuration**  Contains:

- `CONFIG_DAC_DMA_AUTO_16BIT_ALIGN`
- `CONFIG_DAC_ISR_IRAM_SAFE`
- `CONFIG_DAC_ENABLE_DEBUG_LOG`
- `CONFIG_DAC_CTRL_FUNC_IN_IRAM`
- `CONFIG_DAC_SUPPRESS_DEPRECIATE_WARN`

**CONFIG_DAC_CTRL_FUNC_IN_IRAM**

Place DAC control functions into IRAM

*Found in: Component config > Driver Configurations > DAC Configuration*

Place DAC control functions (e.g. `dac_oneshot_output_voltage`) into IRAM, so that this function can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

**Default value:**
- No (disabled)
Chapter 2. API Reference

**CONFIG_DAC_ISR_IRAM_SAFE**

DAC ISR IRAM-Safe

*Found in: Component config > Driver Configurations > DAC Configuration*

Ensure the DAC interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**
- No (disabled)

**CONFIG_DAC.Suppress_Deprecate.Warn**

Suppress legacy driver deprecated warning

*Found in: Component config > Driver Configurations > DAC Configuration*

Whether to suppress the deprecation warnings when using legacy DAC driver (driver/dac.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**
- No (disabled)

**CONFIG_DAC_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > Driver Configurations > DAC Configuration*

Whether to enable the debug log message for DAC driver. Note that, this option only controls the DAC driver log, won’t affect other drivers.

**Default value:**
- No (disabled)

**CONFIG_DAC_DMA_AUTO_16BIT_ALIGN**

Align the continuous data to 16 bit automatically

*Found in: Component config > Driver Configurations > DAC Configuration*

Whether to left shift the continuous data to align every bytes to 16 bits in the driver. On ESP32, although the DAC resolution is only 8 bits, the hardware requires 16 bits data in continuous mode. By enabling this option, the driver will left shift 8 bits for the input data automatically. Only disable this option when you decide to do this step by yourself. Note that the driver will allocate a new piece of memory to save the converted data.

**Default value:**
- Yes (enabled) if SOC_DAC_DMA_16BIT_ALIGN

**USB Serial/JTAG Configuration**

Contains:

- **CONFIG_USJ_NO_AUTO_LS_ON_CONNECTION**

**CONFIG_USJ_NO_AUTO_LS_ON_CONNECTION**

Don’t enter the automatic light sleep when USB Serial/JTAG port is connected

*Found in: Component config > Driver Configurations > USB Serial/JTAG Configuration*

If enabled, the chip will constantly monitor the connection status of the USB Serial/JTAG port. As long as the USB Serial/JTAG is connected, a ESP_PM_NO_LIGHT_SLEEP power management lock will be acquired to prevent the system from entering light sleep. This option can be useful if serial monitoring
is needed via USB Serial/JTAG while power management is enabled, as the USB Serial/JTAG cannot work under light sleep and after waking up from light sleep. Note. This option can only control the automatic Light-Sleep behavior. If esp_light_sleep_start() is called manually from the program, enabling this option will not prevent light sleep entry even if the USB Serial/JTAG is in use.

**Parallel IO Configuration**
Contains:

- **CONFIG_PARLIO_ENABLE_DEBUG_LOG**
- **CONFIG_PARLIO_ISR_IRAM_SAFE**

**CONFIG_PARLIO_ENABLE_DEBUG_LOG**
Enable debug log

*Found in: Component config > Driver Configurations > Parallel IO Configuration*

Whether to enable the debug log message for parallel IO driver. Note that, this option only controls the parallel IO driver log, won’t affect other drivers.

**Default value:**
- No (disabled) if SOC_PARLIO_SUPPORTED

**CONFIG_PARLIO_ISR_IRAM_SAFE**
Parallel IO ISR IRAM-Safe

*Found in: Component config > Driver Configurations > Parallel IO Configuration*

Ensure the Parallel IO interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**
- No (disabled) if SOC_PARLIO_SUPPORTED

**LEDC Configuration**
Contains:

- **CONFIG_LEDC_CTRL_FUNC_IN_IRAM**

**CONFIG_LEDC_CTRL_FUNC_IN_IRAM**
Place LEDC control functions into IRAM

*Found in: Component config > Driver Configurations > LEDC Configuration*

Place LEDC control functions (ledc_update_duty and ledc_stop) into IRAM, so that these functions can be IRAM-safe and able to be called in an IRAM context. Enabling this option can improve driver performance as well.

**Default value:**
- No (disabled)

**I2C Configuration**
Contains:

- **CONFIG_I2C_ENABLE_DEBUG_LOG**
- **CONFIG_I2C_ISR_IRAM_SAFE**
CONFIG_I2C_ISR_IRAM_SAFE

I2C ISR IRAM-Safe

Found in: Component config > Driver Configurations > I2C Configuration

Ensure the I2C interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write). note: This cannot be used in the I2C legacy driver.

Default value:
- No (disabled)

CONFIG_I2C_ENABLE_DEBUG_LOG

Enable I2C debug log

Found in: Component config > Driver Configurations > I2C Configuration

Whether to enable the debug log message for I2C driver. Note that this option only controls the I2C driver log, will not affect other drivers.

note: This cannot be used in the I2C legacy driver.

Default value:
- No (disabled)

eFuse Bit Manager  Contains:

- CONFIG_EFUSE_VIRTUAL
- CONFIG_EFUSE_CUSTOM_TABLE

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.

Default value:
- No (disabled)

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.

Default value:
- “main/esp_efuse_custom_table.csv” if CONFIG_EFUSE_CUSTOM_TABLE

CONFIG_EFUSE_VIRTUAL

Simulate eFuse operations in RAM

Found in: Component config > eFuse Bit Manager

If “n” - No virtual mode. All eFuse operations are real and use eFuse registers. If “y” - The virtual mode is enabled and all eFuse operations (read and write) are redirected to RAM instead of eFuse registers, all permanent changes (via eFuse) are disabled. Log output will state changes that would be applied, but they will not be.

If it is “y”, then SECURE_FLASH_ENCRYPTION_MODE_RELEASE cannot be used. Because the EFUSE VIRT mode is for testing only.
During startup, the eFuses are copied into RAM. This mode is useful for fast tests.

**Default value:**
- No (disabled)

**CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH**

Keep eFuses in flash

*Found in: Component config > eFuse Bit Manager > CONFIG_EFUSE_VIRTUAL*

In addition to the “Simulate eFuse operations in RAM” option, this option just adds a feature to keep eFuses after reboots in flash memory. To use this mode the partition_table should have the *efuse* partition.

partition.csv:
```
o,efuse_em,data,efuse,,0x2000,
p
```

During startup, the eFuses are copied from flash or, in case if flash is empty, from real eFuse to RAM and then update flash. This mode is useful when need to keep changes after reboot (testing secure_boot and flash_encryption).

**CONFIG_EFUSE_VIRTUAL_LOG_ALL_WRITES**

Log all virtual writes

*Found in: Component config > eFuse Bit Manager > CONFIG_EFUSE_VIRTUAL*

If enabled, log efuse burns. This shows changes that would be made.

**ESP-TLS**

Contains:
- **CONFIG_ESP_TLS_INSECURE**
- **CONFIG_ESP_TLS_SERVER_CERT_SELECT_HOOK**
- **CONFIG_ESP_TLS_LIBRARY_CHOOSE**
- **CONFIG_ESP_TLS_CLIENT_SESSION_TICKETS**
- **CONFIG_ESP_DEBUG_WOLFSSL**
- **CONFIG_ESP_TLS_PSK_VERIFICATION**
- **CONFIG_ESP_TLS_SERVER_SESSION_TICKETS**
- **CONFIG_ESP_WOLFSSL_SMALL_CERT_VERIFY**
- **CONFIG_ESP_TLS_SERVER_MIN_AUTH_MODE_OPTIONAL**
- **CONFIG_ESP_TLS_USE_DS_PERIPHERAL**

**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 (CONFIG_ESP_TLS_USING_MBEDTLS)
- wolfSSL (License info in wolfSSL directory README) (CONFIG_ESP_TLS_USING_WOLFSSL)
CONFIG_ESP_TLS_USE_DS_PERIPHERAL

Use Digital Signature (DS) Peripheral with ESP-TLS

*Found in: Component config > ESP-TLS*

Enable use of the Digital Signature Peripheral for ESP-TLS. The DS peripheral can only be used when it is appropriately configured for TLS. Consult the ESP-TLS documentation in ESP-IDF Programming Guide for more details.

**Default value:**
- Yes (enabled)

CONFIG_ESP_TLS_CLIENT_SESSION_TICKETS

Enable client session tickets

*Found in: Component config > ESP-TLS*

Enable session ticket support as specified in RFC5077.

CONFIG_ESP_TLS_SERVER_SESSION_TICKETS

Enable server session tickets

*Found in: Component config > ESP-TLS*

Enable session ticket support as specified in RFC5077.

CONFIG_ESP_TLS_SERVER_SESSION_TICKET_TIMEOUT

Server session ticket timeout in seconds

*Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_SERVER_SESSION_TICKETS*

Sets the session ticket timeout used in the tls server.

**Default value:**
- 86400 if CONFIG_ESP_TLS_SERVER_SESSION_TICKETS

CONFIG_ESP_TLS_SERVER_CERT_SELECT_HOOK

Certificate selection hook

*Found in: Component config > ESP-TLS*

Ability to configure and use a certificate selection callback during server handshake, to select a certificate to present to the client based on the TLS extensions supplied in the client hello (alpn, sni, etc).

CONFIG_ESP_TLS_SERVER_MIN_AUTH_MODEOptional

ESP-TLS Server: Set minimum Certificate Verification mode to Optional

*Found in: Component config > ESP-TLS*

When this option is enabled, the peer (here, the client) certificate is checked by the server, however the handshake continues even if verification failed. By default, the peer certificate is not checked and ignored by the server.

mbedtls_ssl_get_verify_result() can be called after the handshake is complete to retrieve status of verification.
CONFIG_ESP_TLS_PSK_VERIFICATION
Enable PSK verification

*Found in: Component config > ESP-TLS*
Enable support for pre shared key ciphers, supported for both mbedTLS as well as wolfSSL TLS library.

CONFIG_ESP_TLS_INSECURE
Allow potentially insecure options

*Found in: Component config > ESP-TLS*
You can enable some potentially insecure options. These options should only be used for testing purposes. Only enable these options if you are very sure.

CONFIG_ESP_TLS_SKIP_SERVER_CERT_VERIFY
Skip server certificate verification by default (WARNING: ONLY FOR TESTING PURPOSE, READ HELP)

*Found in: Component config > ESP-TLS > CONFIG_ESP_TLS_INSECURE*
After enabling this option the esp-tls client will skip the server certificate verification by default. Note that this option will only modify the default behaviour of esp-tls client regarding server cert verification. The default behaviour should only be applicable when no other option regarding the server cert verification is opted in the esp-tls config (e.g. crt_bundle_attach, use_global_ca_store etc.). WARNING : Enabling this option comes with a potential risk of establishing a TLS connection with a server which has a fake identity, provided that the server certificate is not provided either through API or other mechanism like ca_store etc.

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 upto 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).

**Default value:**
- Yes (enabled) if CONFIG_ESP_TLS_USING_WOLFSSL

CONFIG_ESP_DEBUG_WOLFSSL
Enable debug logs for wolfSSL

*Found in: Component config > ESP-TLS*
Enable detailed debug prints for wolfSSL SSL library.

ADC and ADC Calibration  Contains:
- ADC Calibration Configurations
- CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE
- CONFIG_ADC_DISABLE_DAC_OUTPUT
- CONFIG_ADC_ENABLE_DEBUG_LOG
- CONFIG_ADC_ONESHOT_CTRL_FUNC_IN_IRAM
**CONFIG_ADC_ONESHOT_CTRL_FUNC_IN_IRAM**

Place ISR version ADC oneshot mode read function into IRAM

*Found in: Component config > ADC and ADC Calibration*

Place ISR version ADC oneshot mode read function into IRAM.

**Default value:**
- No (disabled)

**CONFIG_ADC_CONTINUOUS_ISR_IRAM_SAFE**

ADC continuous mode driver ISR IRAM-Safe

*Found in: Component config > ADC and ADC Calibration*

Ensure the ADC continuous mode ISR is IRAM-Safe. When enabled, the ISR handler will be available when the cache is disabled.

**Default value:**
- No (disabled)

**ADC Calibration Configurations**

**CONFIG_ADC_DISABLE_DAC_OUTPUT**

Disable DAC when ADC2 is in use

*Found in: Component config > ADC and ADC Calibration*

By default, this is set. The ADC oneshot driver will disable the output of the corresponding DAC channels: ESP32: IO25 and IO26 ESP32S2: IO17 and IO18

Disable this option so as to measure the output of DAC by internal ADC, for test usage.

**Default value:**
- Yes (enabled)

**CONFIG_ADC_ENABLE_DEBUG_LOG**

Enable ADC debug log

*Found in: Component config > ADC and ADC Calibration*

Whether to enable the debug log message for ADC driver. Note that this option only controls the ADC driver log, will not affect other drivers.

note: This cannot be used in the ADC legacy driver.

**Default value:**
- No (disabled)

**Wireless Coexistence**

Contains:

- `CONFIG_ESP_COEX_EXTERNAL_COEXIST_ENABLE`
- `CONFIG_ESP_COEX_SW_COEXIST_ENABLE`
**CONFIG_ESP_COEX_SW_COEXIST_ENABLE**

Software controls WiFi/Bluetooth coexistence

*Found in: Component config > Wireless Coexistence*

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.

**Default value:**
- Yes (enabled) if `CONFIG_BT_ENABLED`

**CONFIG_ESP_COEX_EXTERNAL_COEXIST_ENABLE**

External Coexistence

*Found in: Component config > Wireless Coexistence*

If enabled, HW External coexistence arbitration is managed by GPIO pins. It can support three types of wired combinations so far which are 1-wired/2-wired/3-wired. User can select GPIO pins in application code with configure interfaces.

This function depends on BT-off because currently we do not support external coex and internal coex simultaneously.

**Common ESP-related**

Contains:
- `CONFIG_ESP_ERR_TO_NAME_LOOKUP`

**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.

**Default value:**
- Yes (enabled)

**ESP-Driver:GPIO Configurations**

Contains:
- `CONFIG_GPIO_CTRL_FUNC_IN_IRAM`

**CONFIG_GPIO_CTRL_FUNC_IN_IRAM**

Place GPIO control functions into IRAM

*Found in: Component config > ESP-Driver:GPIO Configurations*

Place GPIO control functions (like intr_disable/set_level) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context.

**Default value:**
- No (disabled)
ESP-Driver: GPTimer Configurations  Contains:

- `CONFIG_GPTIMER_ENABLE_DEBUG_LOG`
- `CONFIG_GPTIMER_ISR_IRAM_SAFE`
- `CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM`
- `CONFIG_GPTIMER_ISR_HANDLER_IN_IRAM`
- `CONFIG_GPTIMER_SUPPRESS_DEPRECATE_WARN`

**CONFIG_GPTIMER_ISR_HANDLER_IN_IRAM**

Place GPTimer ISR handler into IRAM

*Found in: Component config > ESP-Driver: GPTimer Configurations*

Place GPTimer ISR handler into IRAM for better performance and fewer cache misses.

**Default value:**

- Yes (enabled)

**CONFIG_GPTIMER_CTRL_FUNC_IN_IRAM**

Place GPTimer control functions into IRAM

*Found in: Component config > ESP-Driver: GPTimer Configurations*

Place GPTimer control functions (like start/stop) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

**Default value:**

- No (disabled)

**CONFIG_GPTIMER_ISR_IRAM_SAFE**

GPTimer ISR IRAM-Safe

*Found in: Component config > ESP-Driver: GPTimer Configurations*

Ensure the GPTimer interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**

- No (disabled)

**CONFIG_GPTIMER_SUPPRESS_DEPRECATE_WARN**

Suppress legacy driver deprecated warning

*Found in: Component config > ESP-Driver: GPTimer Configurations*

Whether to suppress the deprecation warnings when using legacy timer group driver (driver/timer.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**

- No (disabled)

**CONFIG_GPTIMER_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > ESP-Driver: GPTimer Configurations*

Whether to enable the debug log message for GPTimer driver. Note that, this option only controls the GPTimer driver log, won’t affect other drivers.
Default value:
- No (disabled)

ESP-Driver: MCPWM Configurations
Contains:
- CONFIG_MCPWM_ENABLE_DEBUG_LOG
- CONFIG_MCPWM_CTRL_FUNC_IN_IRAM
- CONFIG_MCPWM_ISR_IRAM_SAFE
- CONFIG_MCPWM_SUPPRESS_DEPRECATE_WARN

CONFIG_MCPWM_ISR_IRAM_SAFE
Place MCPWM ISR function into IRAM
Found in: Component config > ESP-Driver: MCPWM Configurations
This will ensure the MCPWM interrupt handle is IRAM-Safe, allow to avoid flash cache misses, and also be able to run whilst the cache is disabled. (e.g. SPI Flash write)
Default value:
- No (disabled) if SOC_MCPWM_SUPPORTED

CONFIG_MCPWM_CTRL_FUNC_IN_IRAM
Place MCPWM control functions into IRAM
Found in: Component config > ESP-Driver: MCPWM Configurations
Place MCPWM control functions (like set_compare_value) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.
Default value:
- No (disabled) if SOC_MCPWM_SUPPORTED

CONFIG_MCPWM_SUPPRESS_DEPRECATE_WARN
Suppress legacy driver deprecated warning
Found in: Component config > ESP-Driver: MCPWM Configurations
Whether to suppress the deprecation warnings when using legacy MCPWM driver (driver/mcpwm.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.
Default value:
- No (disabled) if SOC_MCPWM_SUPPORTED

CONFIG_MCPWM_ENABLE_DEBUG_LOG
Enable debug log
Found in: Component config > ESP-Driver: MCPWM Configurations
Whether to enable the debug log message for MCPWM driver. Note that, this option only controls the MCPWM driver log, won’t affect other drivers.
Default value:
- No (disabled) if SOC_MCPWM_SUPPORTED
ESP-Driver:PCNT Configurations Contains:

- `CONFIG_PCNT_ENABLE_DEBUG_LOG`
- `CONFIG_PCNT_ISR_IRAM_SAFE`
- `CONFIG_PCNT_CTRL_FUNC_IN_IRAM`
- `CONFIG_PCNT_SUPPRESS_DEPRECATE_WARN`

**CONFIG_PCNT_CTRL_FUNC_IN_IRAM**

Place PCNT control functions into IRAM

*Found in: Component config > ESP-Driver:PCNT Configurations*

Place PCNT control functions (like start/stop) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

**Default value:**

- No (disabled)

**CONFIG_PCNT_ISR_IRAM_SAFE**

PCNT ISR IRAM-Safe

*Found in: Component config > ESP-Driver:PCNT Configurations*

Ensure the PCNT interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write).

**Default value:**

- No (disabled)

**CONFIG_PCNT_SUPPRESS_DEPRECATE_WARN**

Suppress legacy driver deprecated warning

*Found in: Component config > ESP-Driver:PCNT Configurations*

Whether to suppress the deprecation warnings when using legacy PCNT driver (driver/pcnt.h). If you want to continue using the legacy driver, and don’t want to see related deprecation warnings, you can enable this option.

**Default value:**

- No (disabled)

**CONFIG_PCNT_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > ESP-Driver:PCNT Configurations*

Whether to enable the debug log message for PCNT driver. Note that, this option only controls the PCNT driver log, won’t affect other drivers.

**Default value:**

- No (disabled)

ESP-Driver:SPI Configurations Contains:

- `CONFIG_SPI_MASTER_ISR_IN_IRAM`
- `CONFIG_SPI_SLAVE_ISR_IN_IRAM`
- `CONFIG_SPI_MASTER_IN_IRAM`
- `CONFIG_SPI_SLAVE_IN_IRAM`
Chapter 2. API Reference

**CONFIG_SPI_MASTER_IN_IRAM**

Place transmitting functions of SPI master into IRAM

*Found in: Component config > ESP-Driver:SPI Configurations*

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.

This configuration won’t be available if `CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH` is enabled.

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 > ESP-Driver:SPI Configurations*

Place the SPI master ISR in to IRAM to avoid possible cache miss.

Enabling this configuration is possible only when `HEAP_PLACE_FUNCTION_INTO_FLASH` is disabled since the spi master uses can allocate transactions buffers into DMA memory section using the heap component API that ipso facto has to be placed in IRAM.

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 > ESP-Driver:SPI Configurations*

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.

**Default value:**
- No (disabled)

**CONFIG_SPI_SLAVE_ISR_IN_IRAM**

Place SPI slave ISR function into IRAM

*Found in: Component config > ESP-Driver:SPI Configurations*

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.

**Default value:**
- Yes (enabled)
Chapter 2. API Reference

Ethernet  Contains:
  • CONFIG_ETH_TRANSMIT_MUTEX
  • CONFIG_ETH_USE_OPENETH
  • CONFIG_ETH_USE_SPI_ETHERNET

CONFIG_ETH_USE_SPI_ETHERNET
Support SPI to Ethernet Module

*Found in: Component config > Ethernet*

ESP-IDF can also support some SPI-Ethernet modules.

**Default value:**
  • Yes (enabled)

Contains:
  • CONFIG_ETH_SPI_ETHERNET_DM9051
  • CONFIG_ETH_SPI_ETHERNET_KSZ8851SNL
  • CONFIG_ETH_SPI_ETHERNET_W5500

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. Select this to enable DM9051 driver.

CONFIG_ETH_SPI_ETHERNET_W5500
Use W5500 (MAC RAW)

*Found in: Component config > Ethernet > CONFIG_ETH_USE_SPI_ETHERNET*

W5500 is a HW TCP/IP embedded Ethernet controller. TCP/IP stack, 10/100 Ethernet MAC and PHY are embedded in a single chip. However the driver in ESP-IDF only enables the RAW MAC mode, making it compatible with the software TCP/IP stack. Say yes to enable W5500 driver.

CONFIG_ETH_SPI_ETHERNET_KSZ8851SNL
Use KSZ8851SNL

*Found in: Component config > Ethernet > CONFIG_ETH_USE_SPI_ETHERNET*

The KSZ8851SNL is a single-chip Fast Ethernet controller consisting of a 10/100 physical layer transceiver (PHY), a MAC, and a Serial Peripheral Interface (SPI). Select this to enable KSZ8851SNL 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.

**Default value:**
  • No (disabled)

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.

*Range:*
  • from 1 to 64 if `CONFIG_ETH_USE_OPENETH`

*Default value:*
  • 4 if `CONFIG_ETH_USE_OPENETH`

**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.

*Range:*
  • from 1 to 64 if `CONFIG_ETH_USE_OPENETH`

*Default value:*
  • 1 if `CONFIG_ETH_USE_OPENETH`

**CONFIG_ETH_TRANSMIT_MUTEX**

Enable Transmit Mutex

*Found in: Component config > Ethernet*

Prevents multiple accesses when Ethernet interface is used as shared resource and multiple functionalities might try to access it at a time.

*Default value:*
  • No (disabled)

**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/recieved 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.

*Default value:*
  • No (disabled)
Chapter 2. API Reference

**CONFIG_ESP_EVENT_POST_FROM_ISR**

Support posting events from ISRs

*Found in: Component config > Event Loop Library*

Enable posting events from interrupt handlers.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- Yes (enabled)

**GDB Stub**

Contains:

- `CONFIG_ESP_GDBSTUB_SUPPORT_TASKS`
- `CONFIG_ESP_SYSTEM_GDBSTUB_RUNTIME`

**CONFIG_ESP_SYSTEM_GDBSTUB_RUNTIME**

GDBStub at runtime

*Found in: Component config > GDB Stub*

Enable built-in GDBStub. This allows to debug the target device using serial port: - Run `idf.py monitor` . - Wait for the device to initialize. - Press Ctrl+C to interrupt the execution and enter GDB attached to your device for debugging. NOTE: all UART input will be handled by GDBStub.

**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.

**Default value:**
- 32 if `CONFIG_ESP_GDBSTUB_SUPPORT_TASKS`

**ESP HTTP client**

Contains:

- `CONFIG_ESP_HTTP_CLIENT_ENABLE_BASIC_AUTH`
- `CONFIG_ESP_HTTP_CLIENT_ENABLE_DIGEST_AUTH`
- `CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS`
**CONFIG_ESP_HTTP_CLIENT_ENABLE_HTTPS**

Enable https

*Found in: Component config > ESP HTTP client*

This option will enable https protocol by linking esp-tls library and initializing SSL transport

**Default value:**
- Yes (enabled)

**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

**Default value:**
- No (disabled)

**CONFIG_ESP_HTTP_CLIENT_ENABLE_DIGEST_AUTH**

Enable HTTP Digest Authentication

*Found in: Component config > ESP HTTP client*

This option will enable HTTP Digest Authentication. It is enabled by default, but use of this configuration is not recommended as the password can be derived from the exchange, so it introduces a vulnerability when not using TLS

**Default value:**
- No (disabled)

**HTTP Server**

Contains:
- **CONFIG_HTTPD_QUEUE_WORK_BLOCKING**
- **CONFIG_HTTPD_PURGE_BUF_LEN**
- **CONFIG_HTTPD_LOG_PURGE_DATA**
- **CONFIG_HTTPD_MAX_REQ_HDR_LEN**
- **CONFIG_HTTPD_MAX_URI_LEN**
- **CONFIG_HTTPD_ERR_RESP_NO_DELAY**
- **CONFIG_HTTPD_WS_SUPPORT**

**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

**Default value:**
- 512

**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

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- 32

**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.

**Default value:**
- No (disabled)

**CONFIG_HTTPD_WS_SUPPORT**

WebSocket server support

*Found in: Component config > HTTP Server*

This sets the WebSocket server support.

**Default value:**
- No (disabled)

**CONFIG_HTTPD_QUEUE_WORK_BLOCKING**

httpd_queue_work as blocking API

*Found in: Component config > HTTP Server*

This makes httpd_queue_work() API to wait until a message space is available on UDP control socket. It internally uses a counting semaphore with count set to LWIP_UDP_RECVMBOX_SIZE to achieve this. This config will slightly change API behavior to block until message gets delivered on control socket.
ESP HTTPS OTA  Contains:

- CONFIG_ESP_HTTPS_OTA_ALLOW_HTTP
- CONFIG_ESP_HTTPS_OTA_DECRYPT_CB

CONFIG_ESP_HTTPS_OTA_DECRYPT_CB

Provide decryption callback

*Found in: Component config > ESP HTTPS OTA*

Exposes an additional callback whereby firmware data could be decrypted before being processed by OTA update component. This can help to integrate external encryption related format and removal of such encapsulation layer from firmware image.

**Default value:**

- No (disabled)

CONFIG_ESP_HTTPS_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

**Default value:**

- No (disabled)

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

Hardware Settings  Contains:

- Chip revision
- Crypto DPA Protection
- ESP_SLEEP_WORKAROUND
- ETM Configuration
- GDMA Configuration
- MAC Config
- Main XTAL Config
- Peripheral Control
- RTC Clock Config
- Sleep Config

Chip revision  Contains:

- CONFIG_ESP_REV_NEW_CHIP_TEST
- CONFIG_ESP32S2_REV_MIN
**CONFIG_ESP32S2_REV_MIN**

Minimum Supported ESP32-S2 Revision

*Found in: Component config > Hardware Settings > Chip revision*

Required minimum chip revision. ESP-IDF will check for it and reject to boot if the chip revision fails the check. This ensures the chip used will have some modifications (features, or bugfixes). The compiled binary will only support chips above this revision, this will also help to reduce binary size.

Available options:

- Rev v0.0 (ECO0) (CONFIG_ESP32S2_REV_MIN_0)
- Rev v1.0 (ECO1) (CONFIG_ESP32S2_REV_MIN_1)

**CONFIG_ESP_REV_NEWCHIP_TEST**

Internal test mode

*Found in: Component config > Hardware Settings > Chip revision*

For internal chip testing, a small number of new versions chips didn’t update the version field in eFuse, you can enable this option to force the software recognize the chip version based on the rev selected in menuconfig.

**Default value:**

- No (disabled)

**MAC Config** Contains:

- **CONFIG_ESP32S2_UNIVERSAL_MAC_ADDRESSES**

**CONFIG_ESP32S2_UNIVERSAL_MAC_ADDRESSES**

Number of universally administered (by IEEE) MAC address

*Found in: Component config > Hardware Settings > MAC Config*

Configure the number of universally administered (by IEEE) MAC addresses. During initialization, MAC addresses for each network interface are generated or derived from a single base MAC address. If the number of universal MAC addresses is Two, all interfaces (WiFi station, WiFi softap) receive a universally administered MAC address. They are generated sequentially by adding 0, and 1 (respectively) to the final octet of the base MAC address. If the number of universal MAC addresses is one, only WiFi station receives a universally administered MAC address. It’s generated by adding 0 to the base MAC address. The WiFi softap receives local MAC addresses. It’s derived from the universal WiFi station MAC addresses. 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 1 or 2 per device.)

Available options:

- One (CONFIG_ESP32S2_UNIVERSAL_MAC_ADDRESSES_ONE)
- Two (CONFIG_ESP32S2_UNIVERSAL_MAC_ADDRESSES_TWO)

**Sleep Config** Contains:

- **CONFIG_ESP_SLEEP_GPIO_ENABLE_INTERNAL_RESISTORS**
- **CONFIG_ESP_SLEEP_CACHE_SAFE_ASSERTION**
- **CONFIG_ESP_SLEEP_EVENT_CALLBACKS**
- **CONFIG_ESP_SLEEP_DEBUG**
CONFIG_ESP_SLEEP_POWER_DOWN_FLASH

Power down flash in light sleep when there is no SPIRAM

Found in: Component config > Hardware Settings > Sleep Config

If enabled, chip will try to power down flash as part of esp_light_sleep_start(), which costs more time when chip wakes up. Can only be enabled if there is no SPIRAM configured.

This option will power down flash under a strict but relatively safe condition. Also, it is possible to power down flash under a relaxed condition by using esp_sleep_pd_config() to set ESP_PD_DOMAIN_VDDSIO to ESP_PD_OPTION_OFF. It should be noted that there is a risk in powering down flash, you can refer ESP-IDF Programming Guide/API Reference/System API/Sleep Modes/Power-down of Flash for more details.

CONFIG_ESP_SLEEP_FLASH_LEAKAGE_WORKAROUND

Pull-up Flash CS pin in light sleep

Found in: Component config > Hardware Settings > Sleep Config

All IOs will be set to isolate(floating) state by default during sleep. Since the power supply of SPI Flash is not lost during lightsleep, if its CS pin is recognized as low level(selected state) in the floating state, there will be a large current leakage, and the data in Flash may be corrupted by random signals on other SPI pins. Select this option will set the CS pin of Flash to PULL-UP state during sleep, but this will increase the sleep current about 10 uA. If you are developing with esp32xx modules, you must select this option, but if you are developing with chips, you can also pull up the CS pin of SPI Flash in the external circuit to save power consumption caused by internal pull-up during sleep. (!!! Don’t deselect this option if you don’t have external SPI Flash CS pin pullups.)

CONFIG_ESP_SLEEP_PSRAM_LEAKAGE_WORKAROUND

Pull-up PSRAM CS pin in light sleep

Found in: Component config > Hardware Settings > Sleep Config

All IOs will be set to isolate(floating) state by default during sleep. Since the power supply of PSRAM is not lost during lightsleep, if its CS pin is recognized as low level(selected state) in the floating state, there will be a large current leakage, and the data in PSRAM may be corrupted by random signals on other SPI pins. Select this option will set the CS pin of PSRAM to PULL-UP state during sleep, but this will increase the sleep current about 10 uA. If you are developing with esp32xx modules, you must select this option, but if you are developing with chips, you can also pull up the CS pin of PSRAM in the external circuit to save power consumption caused by internal pull-up during sleep. (!!! Don’t deselect this option if you don’t have external PSRAM CS pin pullups.)

Default value:
  • Yes (enabled) if CONFIG_SPIRAM

CONFIG_ESP_SLEEP_MSPI_NEED_ALL_IO_PU

Pull-up all SPI pins in light sleep

Found in: Component config > Hardware Settings > Sleep Config
To reduce leakage current, some types of SPI Flash/RAM only need to pull up the CS pin during light sleep. But there are also some kinds of SPI Flash/RAM that need to pull up all pins. It depends on the SPI Flash/RAM chip used.

**CONFIG_ESP_SLEEP_GPIO_RESET_WORKAROUND**

light sleep GPIO reset workaround

*Found in: Component config > Hardware Settings > Sleep Config*

esp32c2, esp32c3, esp32s3, esp32c6 and esp32h2 will reset at wake-up if GPIO is received a small electrostatic pulse during light sleep, with specific condition

- GPIO needs to be configured as input-mode only
- The pin receives a small electrostatic pulse, and reset occurs when the pulse voltage is higher than 6 V

For GPIO set to input mode only, it is not a good practice to leave it open/floating. The hardware design needs to controlled it with determined supply or ground voltage is necessary.

This option provides a software workaround for this issue. Configure to isolate all GPIO pins in sleep state.

**CONFIG_ESP_SLEEP_CACHE_SAFE_ASSERTION**

Check the cache safety of the sleep wakeup code in sleep process

*Found in: Component config > Hardware Settings > Sleep Config*

Enabling it will check the cache safety of the code before the flash power is ready after light sleep wakeup, and check PM_SLP_IRAM_OPT related code cache safety. This option is only for code quality inspection. Enabling it will increase the time overhead of entering and exiting sleep. It is not recommended to enable it in the release version.

**Default value:**
- No (disabled)

**CONFIG_ESP_SLEEP_DEBUG**

esp sleep debug

*Found in: Component config > Hardware Settings > Sleep Config*

Enable esp sleep debug.

**Default value:**
- No (disabled)

**CONFIG_ESP_SLEEP_GPIO_ENABLE_INTERNAL_RESISTORS**

Allow to enable internal pull-up/downs for the Deep-Sleep wakeup IOs

*Found in: Component config > Hardware Settings > Sleep Config*

When using rtc gpio wakeup source during deepsleep without external pull-up/downs, you may want to make use of the internal ones.

**Default value:**
- Yes (enabled)
CONFIG_ESP_SLEEP_EVENT_CALLBACKS

Enable registration of sleep event callbacks

*Found in:* Component config > Hardware Settings > Sleep Config

If enabled, it allows user to register sleep event callbacks. It is primarily designed for internal developers and customers can use PM_LIGHT_SLEEP_CALLBACKS as an alternative.

**NOTE:** These callbacks are executed from the IDLE task context hence you cannot have any blocking calls in your callbacks.

**NOTE:** Enabling these callbacks may change sleep duration calculations based on time spent in callback and hence it is highly recommended to keep them as short as possible.

**Default value:**
- No (disabled) if CONFIG_FREERTOS_USE_TICKLESS_IDLE

ESP_SLEEP_WORKAROUND

RTC Clock Config  
Contains:
- CONFIG_RTC_XTAL_CAL_RETRY
- CONFIG_RTC_CLK_CAL_CYCLES
- CONFIG_RTC_CLK_SRC

CONFIG_RTC_CLK_SRC

RTC clock source

*Found in:* Component config > Hardware Settings > RTC Clock Config

Choose which clock is used as RTC clock source.

- **“Internal 90kHz 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 < Vcm < 0.5Vamp, where Vamp is the signal amplitude.**
- **“Internal 8MHz oscillator divided by 256” option results in higher deep sleep current (by 5uA) but has better frequency stability than the internal 90kHz oscillator. It does not require external components.**

Available options:

- Internal 90kHz RC oscillator (CONFIG_RTC_CLK_SRC_INT_RC)
- External 32kHz crystal (CONFIG_RTC_CLK_SRC_EXT_CRYSTAL)
- External 32kHz oscillator at 32K_XN pin (CONFIG_RTC_CLK_SRC_EXT_OSC)
- Internal 8MHz oscillator, divided by 256 (~32kHz) (CONFIG_RTC_CLK_SRC_INT_8MD256)

CONFIG_RTC_CLK_CAL_CYCLES

Number of cycles for RTC_SLOW_CLK calibration

*Found in:* Component config > Hardware Settings > RTC Clock Config
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:

- 90000 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.

**Range:**
- from 0 to 125000

**Default value:**
- 3000 if `CONFIG_RTC_CLK_SRC_EXT_CRYSTAL` || `CONFIG_RTC_CLK_SRC_EXT_OSC` || `CONFIG_RTC_CLK_SRC_INT_8MD256`
- 576

**CONFIG_RTC_XTAL_CAL_RETRY**

Number of attempts to repeat 32k XTAL calibration

*Found in: Component config > Hardware Settings > RTC Clock Config*

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.

**Default value:**
- 3 if `CONFIG_RTC_CLK_SRC_EXT_CRYSTAL`

**Peripheral Control**

Contains:

- `CONFIG_PERIPH_CTRL_FUNC_IN_IRAM`

**CONFIG_PERIPH_CTRL_FUNC_IN_IRAM**

Place peripheral control functions into IRAM

*Found in: Component config > Hardware Settings > Peripheral Control*

Place peripheral control functions (e.g. `periph_module_reset`) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context.

**Default value:**
- No (disabled)

**ETM Configuration**

Contains:

- `CONFIG_ETM_ENABLE_DEBUG_LOG`

**CONFIG_ETM_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > Hardware Settings > ETM Configuration*

Wether to enable the debug log message for ETM core driver. Note that, this option only controls the ETM related driver log, won’t affect other drivers.

**Default value:**
GDMA Configuration

Contains:

- `CONFIG_GDMA_ENABLE_DEBUG_LOG`
- `CONFIG_GDMA_ISR_IRAM_SAFE`
- `CONFIG_GDMA_CTRL_FUNC_IN_IRAM`

**CONFIG_GDMA_CTRL_FUNC_IN_IRAM**

Place GDMA control functions into IRAM

*Found in: Component config > Hardware Settings > GDMA Configuration*

Place GDMA control functions (like start/stop/append/reset) into IRAM, so that these functions can be IRAM-safe and able to be called in the other IRAM interrupt context. Enabling this option can improve driver performance as well.

**Default value:**

- No (disabled) if SOC_GDMA_SUPPORTED

**CONFIG_GDMA_ISR_IRAM_SAFE**

GDMA ISR IRAM-Safe

*Found in: Component config > Hardware Settings > GDMA Configuration*

This will ensure the GDMA interrupt handler is IRAM-Safe, allow to avoid flash cache misses, and also be able to run whilst the cache is disabled. (e.g. SPI Flash write).

**Default value:**

- No (disabled) if SOC_GDMA_SUPPORTED

**CONFIG_GDMA_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > Hardware Settings > GDMA Configuration*

Wether to enable the debug log message for GDMA driver. Note that, this option only controls the GDMA driver log, won’t affect other drivers.

**Default value:**

- No (disabled) if SOC_GDMA_SUPPORTED

**Main XTAL Config**

Contains:

- `CONFIG_XTAL_FREQ_SEL`

**CONFIG_XTAL_FREQ_SEL**

Main XTAL frequency

*Found in: Component config > Hardware Settings > Main XTAL Config*

This option selects the operating frequency of the XTAL (crystal) clock used to drive the ESP target. The selected value MUST reflect the frequency of the given hardware.

Note: The XTAL_FREQ_AUTO option allows the ESP target to automatically estimating XTAL clock’s operating frequency. However, this feature is only supported on the ESP32. The ESP32 uses the internal 8MHZ as a reference when estimating. Due to the internal oscillator’s frequency being temperature dependent, usage of the XTAL_FREQ_AUTO is not recommended in applications that operate in high ambient temperatures or use high-temperature qualified chips and modules.
Available options:

- 24 MHz (CONFIG_XTAL_FREQ_24)
- 26 MHz (CONFIG_XTAL_FREQ_26)
- 32 MHz (CONFIG_XTAL_FREQ_32)
- 40 MHz (CONFIG_XTAL_FREQ_40)
- Autodetect (CONFIG_XTAL_FREQ_AUTO)

**Crypto DPA Protection**  Contains:

- `CONFIG_ESP_CRYPTO_DPA_PROTECTION_AT_STARTUP`

**CONFIG_ESP_CRYPTO_DPA_PROTECTION_AT_STARTUP**

Enable crypto DPA protection at startup

*Found in: Component config > Hardware Settings > Crypto DPA Protection*

This config controls the DPA (Differential Power Analysis) protection knob for the crypto peripherals. DPA protection dynamically adjusts the clock frequency of the crypto peripheral. DPA protection helps to make it difficult to perform SCA attacks on the crypto peripherals. However, there is also associated performance impact based on the security level set. Please refer to the TRM for more details.

**Default value:**

- Yes (enabled) if SOC_CRYPTO_DPA_PROTECTION_SUPPORTED

**CONFIG_ESP_CRYPTO_DPA_PROTECTION_LEVEL**

DPA protection level

*Found in: Component config > Hardware Settings > Crypto DPA Protection > CONFIG_ESP_CRYPTO_DPA_PROTECTION_AT_STARTUP*

Configure the DPA protection security level

Available options:

- Security level low (CONFIG_ESP_CRYPTO_DPA_PROTECTION_LEVEL_LOW)
- Security level medium (CONFIG_ESP_CRYPTO_DPA_PROTECTION_LEVEL_MEDIUM)
- Security level high (CONFIG_ESP_CRYPTO_DPA_PROTECTION_LEVEL_HIGH)

**LCD and Touch Panel**  Contains:

- `LCD Peripheral Configuration`

**LCD Peripheral Configuration**  Contains:

- `CONFIG_LCD_ENABLE_DEBUG_LOG`
- `CONFIG_LCD_PANEL_IO_FORMAT_BUF_SIZE`
- `CONFIG_LCD_RGB_RESTART_IN_VSYNC`
- `CONFIG_LCD_RGB_ISR_IRAM_SAFE`
Chapter 2. API Reference

**CONFIG_LCD_PANEL_IO_FORMAT_BUF_SIZE**

LCD panel io format buffer size

*Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration*

LCD driver allocates an internal buffer to transform the data into a proper format, because of the endian order mismatch. This option is to set the size of the buffer, in bytes.

**Default value:**
- 32

**CONFIG_LCD_ENABLE_DEBUG_LOG**

Enable debug log

*Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration*

Whether to enable the debug log message for LCD driver. Note that, this option only controls the LCD driver log, won’t affect other drivers.

**Default value:**
- No (disabled)

**CONFIG_LCD_RGB_ISR_IRAM_SAFE**

RGB LCD ISR IRAM-Safe

*Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration*

Ensure the LCD interrupt is IRAM-Safe by allowing the interrupt handler to be executable when the cache is disabled (e.g. SPI Flash write). If you want the LCD driver to keep flushing the screen even when cache ops disabled, you can enable this option. Note, this will also increase the IRAM usage.

**Default value:**
- No (disabled) if SOC_LCD_RGB_SUPPORTED

**CONFIG_LCD_RGB_RESTART_IN_VSYNC**

Restart transmission in VSYNC

*Found in: Component config > LCD and Touch Panel > LCD Peripheral Configuration*

Reset the GDMA channel every VBlank to stop permanent desyncs from happening. Only need to enable it when in your application, the DMA can’t deliver data as fast as the LCD consumes it.

**Default value:**
- No (disabled) if SOC_LCD_RGB_SUPPORTED

**ESP NETIF Adapter**

Contains:

- **CONFIG_ESP_NETIF_BRIDGE_EN**
- **CONFIG_ESP_NETIF_L2_TAP**
- **CONFIG_ESP_NETIF_IP_LOST_TIMER_INTERVAL**
- **CONFIG_ESP_NETIF_USE_TCP/IP_STACK_LIB**
- **CONFIG_ESP_NETIF_RECEIVE REPORT_ERRORS**

**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.

**Range:**
- from 0 to 65535

**Default value:**
- 120

**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 (CONFIG_ESP_NETIF_TCPIP_LWIP)
  lwIP is a small independent implementation of the TCP/IP protocol suite.
- Loopback (CONFIG_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_RECEIVE_REPORT_ERRORS**

Use esp_err_t to report errors from esp_netif_receive

*Found in: Component config > ESP NETIF Adapter*

Enable if esp_netif_receive() should return error code. This is useful to inform upper layers that packet input to TCP/IP stack failed, so the upper layers could implement flow control. This option is disabled by default due to backward compatibility and will be enabled in v6.0 (IDF-7194)

**Default value:**
- No (disabled)

**CONFIG_ESP_NETIF_L2_TAP**

Enable netif L2 TAP support

*Found in: Component config > ESP NETIF Adapter*

A user program can read/write link layer (L2) frames from/to ESP TAP device. The ESP TAP device can be currently associated only with Ethernet physical interfaces.

**CONFIG_ESP_NETIF_L2_TAP_MAX_FDS**

Maximum number of opened L2 TAP File descriptors

*Found in: Component config > ESP NETIF Adapter > CONFIG_ESP_NETIF_L2_TAP*

Maximum number of opened File descriptors (FD’s) associated with ESP TAP device. ESP TAP FD’s take up a certain amount of memory, and allowing fewer FD’s to be opened at the same time conserves memory.

**Range:**
- from 1 to 10 if CONFIG_ESP_NETIF_L2_TAP

**Default value:**
- 5 if CONFIG_ESP_NETIF_L2_TAP
**CONFIG_ESP_NETIF_L2_TAP_RX_QUEUE_SIZE**

Size of L2 TAP Rx queue

*Found in: Component config > ESP NETIF Adapter > CONFIG_ESP_NETIF_L2_TAP*

Maximum number of frames queued in opened File descriptor. Once the queue is full, the newly arriving frames are dropped until the queue has enough room to accept incoming traffic (Tail Drop queue management).

**Range:**
- from 1 to 100 if `CONFIG_ESP_NETIF_L2_TAP`

**Default value:**
- 20 if `CONFIG_ESP_NETIF_L2_TAP`

**CONFIG_ESP_NETIF_BRIDGE_EN**

Enable LwIP IEEE 802.1D bridge

*Found in: Component config > ESP NETIF Adapter*

Enable LwIP IEEE 802.1D bridge support in ESP-NETIF. Note that “Number of clients store data in netif” (LWIP_NUM_NETIF_CLIENT_DATA) option needs to be properly configured to be LwIP bridge available!

**Default value:**
- No (disabled)

**Partition API Configuration**

**PHY**

Contains:

- `CONFIG_ESP_PHY_CALIBRATION_MODE`
- `CONFIG_ESP_PHY_ENABLE_USB`
- `CONFIG_ESP_PHY_IMPROVE_RX_11B`
- `CONFIG_ESP_PHY_MAX_WIFI_TX_POWER`
- `CONFIG_ESP_PHY_MAC_BB_PD`
- `CONFIG_ESP_PHY_REDUCE_TX_POWER`
- `CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE`
- `CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION`

**CONFIG_ESP_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’.

**Default value:**
- Yes (enabled)
### CONFIG_ESP_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.

*Default value:*
- No (disabled)

*Contains:*
- **CONFIG_ESP_PHY_DEFAULT_INIT_IF_INVALID**
- **CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN**

### CONFIG_ESP_PHY_DEFAULT_INIT_IF_INVALID

Reset default PHY init data if invalid

*Found in: Component config > PHY > CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

If enabled, PHY init data will be restored to default if it cannot be verified successfully to avoid endless bootloops.

If unsure, choose ‘n’.

*Default value:*
- No (disabled) if **CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION**

### CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN

Support multiple PHY init data bin

*Found in: Component config > PHY > CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION*

If enabled, the corresponding PHY init data type can be automatically switched according to the country code. China’s PHY init data bin is used by default. Can be modified by country information in API `esp_wifi_set_country()`. The priority of switching the PHY init data type is: 1. Country configured by API `esp_wifi_set_country()` and the parameter policy is WIFI_COUNTRY_POLICY_MANUAL. 2. Country notified by the connected AP. 3. Country configured by API `esp_wifi_set_country()` and the parameter policy is WIFI_COUNTRY_POLICY_AUTO.

*Default value:*
- No (disabled) if **CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION** ***&&*** **CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN_EMBED**

### CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN_EMBED

Support embedded multiple phy init data bin to app bin

*Found in: Component config > PHY > CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION > CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN***

If enabled, multiple phy init data bin will be embedded into app bin. If not enabled, multiple phy init data bin will still leave alone, and need to be flashed by users.

*Default value:*
- No (disabled) if **CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN** ***&&*** **CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION**
CONFIG_ESP_PHY_INIT_DATA_ERROR

Terminate operation when PHY init data error

*Found in: Component config > PHY > CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION > CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN*

If enabled, when an error occurs while the PHY init data is updated, the program will terminate and restart. If not enabled, the PHY init data will not be updated when an error occurs.

**Default value:**
- No (disabled) if CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN && CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION

CONFIG_ESP_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.

**Range:**
- from 10 to 20

**Default value:**
- 20

CONFIG_ESP_PHY_MAC_BB_PD

Power down MAC and baseband of Wi-Fi and Bluetooth when PHY is disabled

*Found in: Component config > PHY*

If enabled, the MAC and baseband of Wi-Fi and Bluetooth will be powered down when PHY is disabled. Enabling this setting reduces power consumption by a small amount but increases RAM use by approximately 4 KB (Wi-Fi only), 2 KB (Bluetooth only) or 5.3 KB (Wi-Fi + Bluetooth).

**Default value:**
- No (disabled) if SOC_PM_SUPPORT_MAC_BB_PD && CONFIG_FREERTOS_USE_TICKLESS_IDLE

CONFIG_ESP_PHY_REDUCE_TX_POWER

Reduce PHY TX power when brownout reset

*Found in: Component config > PHY*

When brownout reset occurs, reduce PHY TX power to keep the code running.

**Default value:**
- No (disabled)

CONFIG_ESP_PHY_ENABLE_USB

Enable USB when phy init

*Found in: Component config > PHY*

When using USB Serial/JTAG/OTG/CDC, PHY should enable USB, otherwise USB module can not work properly. Notice: Enabling this configuration option will slightly impact wifi performance.

**Default value:**
- No (disabled)
CONFIG_ESP_PHY_CALIBRATION_MODE

Calibration mode

Found in: Component config > PHY

Select PHY calibration mode. During RF initialization, the partial calibration method is used by default for RF calibration. Full calibration takes about 100ms more than partial calibration. If boot duration is not critical, it is suggested to use the full calibration method. No calibration method is only used when the device wakes up from deep sleep.

Available options:

- Calibration partial (CONFIG_ESP_PHY_RF_CAL_PARTIAL)
- Calibration none (CONFIG_ESP_PHY_RF_CAL_NONE)
- Calibration full (CONFIG_ESP_PHY_RF_CAL_FULL)

CONFIG_ESP_PHY_IMPROVE_RX_11B

Improve Wi-Fi receive 11b pkts

Found in: Component config > PHY

This is a workaround to improve Wi-Fi receive 11b pkts for some modules using AC-DC power supply with high interference, enable this option will sacrifice Wi-Fi OFDM receive performance. But to guarantee 11b receive performance serves as a bottom line in this case.

Default value:
- No (disabled) if SOC_PHY_IMPROVE_RX_11B

Power Management  Contains:

- CONFIG_PM_LIGHTSLEEP_RTC_OSC_CAL_INTERVAL
- CONFIG_PM_SLP_DISABLE_GPIO
- CONFIG_PM_LIGHT_SLEEP_CALLBACKS
- CONFIG_PM_POWER_DOWN_CPU_IN_LIGHT_SLEEP
- CONFIG_PM_POWER_DOWN_PERIPHERAL_IN_LIGHT_SLEEP
- CONFIG_PM_SLP_IRAM_OPT
- CONFIG_PM_RTOS_IDLE_OPT
- 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.

Default value:
- No (disabled) if __DOXYGEN__

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 DEF-
FAULT_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.

Default value:
• No (disabled) if CONFIG_PM_ENABLE

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.

Default value:
• No (disabled) if CONFIG_PM_ENABLE

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.

Default value:
• No (disabled) if CONFIG_PM_ENABLE

CONFIG_PM_SLP_IRAM_OPT
Put lightsleep related codes in internal RAM

Found in: Component config > Power Management

If enabled, about 2.1KB of lightsleep related source code would be in IRAM and chip would sleep longer 
for 310us at 160MHz CPU frequency most each time. This feature is intended to be used when lower 
power consumption is needed while there is enough place in IRAM to place source code.

CONFIG_PM_RTOS_IDLE_OPT
Put RTOS IDLE related codes in internal RAM

Found in: Component config > Power Management

If enabled, about 180Bytes of RTOS_IDLE related source code would be in IRAM and chip would sleep 
longer for 20us at 160MHz CPU frequency most each time. This feature is intended to be used when lower 
power consumption is needed while there is enough place in IRAM to place source code.

CONFIG_PM_SLP_DISABLE_GPIO
Disable all GPIO when chip at sleep

Found in: Component config > Power Management
This feature is intended to disable all GPIO pins at automatic sleep to get a lower power mode. If enabled, chips will disable all GPIO pins at automatic sleep to reduce about 200~300 uA current. If you want to specifically use some pins normally as chip wakes when chip sleeps, you can call `gpio_sleep_sel_dis` to disable this feature on those pins. You can also keep this feature on and call `gpio_sleep_set_direction` and `gpio_sleep_set_pull_mode` to have a different GPIO configuration at sleep. Warning: If you want to enable this option on ESP32, you should enable `GPIO_ESP32_SUPPORT_SWITCH_SLP_PULL` at first, otherwise you will not be able to switch pullup/pulldown mode.

**CONFIG_PM_LIGHTSLEEP_RTC_OSC_CAL_INTERVAL**

Calibrate the RTC_FAST/SLOW clock every N times of light sleep

*Found in: Component config > Power Management*

The value of this option determines the calibration interval of the RTC_FAST/SLOW clock during sleep when power management is enabled. When it is configured as N, the RTC_FAST/SLOW clock will be calibrated every N times of lightsleep. Decreasing this value will increase the time the chip is in the active state, thereby increasing the average power consumption of the chip. Increasing this value can reduce the average power consumption, but when the external environment changes drastically and the chip RTC_FAST/SLOW oscillator frequency drifts, it may cause system instability.

*Range:*
- from 1 to 128 if `CONFIG_PM_ENABLE`

*Default value:*
- 1 if `CONFIG_PM_ENABLE`

**CONFIG_PM_POWER_DOWN_CPU_IN_LIGHT_SLEEP**

Power down CPU in light sleep

*Found in: Component config > Power Management*

If enabled, the CPU will be powered down in light sleep, ESP chips supports saving and restoring CPU’s running context before and after light sleep, the feature provides applications with seamless CPU powered-down lightsleep without user awareness. But this will takes up some internal memory. On esp32c3 soc, enabling this option will consume 1.68 KB of internal RAM and will reduce sleep current consumption by about 100 uA. On esp32s3 soc, enabling this option will consume 8.58 KB of internal RAM and will reduce sleep current consumption by about 650 uA.

*Default value:*
- Yes (enabled) if SOC_PM_SUPPORT_CPU_PD

**CONFIG_PM_POWER_DOWN_PERIPHERAL_IN_LIGHT_SLEEP**

Power down Digital Peripheral in light sleep (EXPERIMENTAL)

*Found in: Component config > Power Management*

If enabled, digital peripherals will be powered down in light sleep, it will reduce sleep current consumption by about 100 uA. Chip will save/restore register context at sleep/wake time to keep the system running. Enabling this option will increase static RAM and heap usage, the actual cost depends on the peripherals you have initialized. In order to save/restore the context of the necessary hardware for FreeRTOS to run, it will need at least 4.55 KB free heap at sleep time. Otherwise sleep will not power down the peripherals.

Note1: Please use this option with caution, the current IDF does not support the retention of all peripherals. When the digital peripherals are powered off and a sleep and wake-up is completed, the peripherals that have not saved the running context are equivalent to performing a reset. !!! Please confirm the peripherals used in your application and their sleep retention support status before enabling this option, peripherals sleep retention driver support status is tracked in power_management.rst
Chapter 2. API Reference

Note 2: When this option is enabled simultaneously with `FREERTOS_USE_TICKLESS_IDLE`, since the UART will be powered down, the uart FIFO will be flushed before sleep to avoid data loss, however, this has the potential to block the sleep process and cause the wakeup time to be skipped, which will cause the tick of freertos to not be compensated correctly when returning from sleep and cause the system to crash. To avoid this, you can increase `FREERTOS_IDLE_TIME_BEFORE_SLEEP` threshold in menuconfig.

**Default value:**
- No (disabled) if `SOC_PAU_SUPPORTED`

### CONFIG_PM_LIGHT_SLEEP_CALLBACKS

Enable registration of pm light sleep callbacks

*Found in: Component config > Power Management*

If enabled, it allows user to register entry and exit callbacks which are called before and after entering auto light sleep.

**NOTE:** These callbacks are executed from the IDLE task context hence you cannot have any blocking calls in your callbacks.

**NOTE:** Enabling these callbacks may change sleep duration calculations based on time spent in callback and hence it is highly recommended to keep them as short as possible

**Default value:**
- No (disabled) if `CONFIG_FREERTOS_USE_TICKLESS_IDLE`

### ESP PSRAM

Contains:
- `CONFIG_SPIRAM`

### CONFIG_SPIRAM

Support for external, SPI-connected RAM

*Found in: Component config > ESP PSRAM*

This enables support for an external SPI RAM chip, connected in parallel with the main SPI flash chip.

### SPI RAM config

Contains:
- `CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY`
- `CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY`
- `CONFIG_SPIRAM_BOOT_INIT`
- `CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL`
- `CONFIG_SPIRAM_FETCH_INSTRUCTIONS`
- `CONFIG_SPIRAM_RODATA`
- `CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL`
- `CONFIG_SPIRAM_MEMTEST`
- `CONFIG_SPIRAM_SPEED`
- `CONFIG_SPIRAM_USE`
- `CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP`
- `CONFIG_SPIRAM_TYPE`

### CONFIG_SPIRAM_TYPE

Type of SPI RAM chip in use

*Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config*

Available options:
• Auto-detect (CONFIG_SPIRAM_TYPE_AUTO)
• ESP-PSRAM16 or APS1604 (CONFIG_SPIRAM_TYPE_ESPPSRAM16)
• ESP-PSRAM32 (CONFIG_SPIRAM_TYPE_ESPPSRAM32)
• ESP-PSRAM64 or LY68L6400 (CONFIG_SPIRAM_TYPE_ESPPSRAM64)

**CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY**

Allow external memory as an argument to xTaskCreateStatic

*Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config*

Accessing memory in SPIRAM has certain restrictions, so task stacks allocated by xTaskCreate are by default allocated from internal RAM.

This option allows for passing memory allocated from SPIRAM to be passed to xTaskCreateStatic. This should only be used for tasks where the stack is never accessed while the cache is disabled.

**CONFIG_SPIRAM_FETCH_INSTRUCTIONS**

Move Instructions in Flash to PSRAM

*Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config*

If enabled, instructions in flash will be moved into PSRAM on startup. If SPIRAM_RODATA is also enabled, code that requires execution during an SPI1 Flash operation can forgo being placed in IRAM, thus optimizing RAM usage (see External RAM documentation for more details).

**CONFIG_SPIRAM_RODATA**

Move Read-Only Data in Flash to PSRAM

*Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config*

If enabled, rodata in flash will be moved into PSRAM on startup. If SPIRAM_FETCH_INSTRUCTIONS is also enabled, code that requires execution during an SPI1 Flash operation can forgo being placed in IRAM, thus optimizing RAM usage (see External RAM documentation for more details).

**CONFIG_SPIRAM_SPEED**

Set RAM clock speed

*Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config*

Select the speed for the SPI RAM chip.

Available options:

• 80MHz clock speed (CONFIG_SPIRAM_SPEED_80M)
• 40Mhz clock speed (CONFIG_SPIRAM_SPEED_40M)
• 26Mhz clock speed (CONFIG_SPIRAM_SPEED_26M)
• 20Mhz clock speed (CONFIG_SPIRAM_SPEED_20M)

**CONFIG_SPIRAM_BOOT_INIT**

Initialize SPI RAM during startup

*Found in: Component config > ESP PSRAM > CONFIG_SPIRAM > 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 > ESP PSRAM > CONFIG_SPIRAM > 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 > ESP PSRAM > CONFIG_SPIRAM > 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 (CONFIG_SPIRAM_USE_MEMMAP)
- Make RAM allocatable using heap_caps_malloc(..., MALLOC_CAP_SPIRAM) (CONFIG_SPIRAM_USE_CAPS_ALLOC)
- Make RAM allocatable using malloc() as well (CONFIG_SPIRAM_USE_MALLOC)

CONFIG_SPIRAM_MEMTEST

Run memory test on SPI RAM initialization

*Found in:* Component config > ESP PSRAM > CONFIG_SPIRAM > 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 > ESP PSRAM > CONFIG_SPIRAM > 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 > ESP PSRAM > CONFIG_SPIRAM > 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 > ESP PSRAM > CONFIG_SPIRAM > 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 > ESP PSRAM > CONFIG_SPIRAM > SPI RAM config*

If enabled, variables with EXT_RAM_BSS_ATTR attribute will be placed in SPIRAM instead of internal DRAM. BSS section of lwip, net80211, pp, br libraries will be automatically placed in SPIRAM. BSS sections from other object files and libraries can also be placed in SPIRAM through linker fragment scheme extram_bss.

Note that the variables placed in SPIRAM using EXT_RAM_BSS_ATTR will be zero initialized.

ESP Ringbuf  Contains:

- CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH

CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH

Place non-ISR ringbuf functions into flash

*Found in: Component config > ESP Ringbuf*

Place non-ISR ringbuf functions (like xRingbufferCreate/xRingbufferSend) into flash. This frees up IRAM, but the functions can no longer be called when the cache is disabled.

**Default value:**

- No (disabled)

CONFIG_RINGBUF_PLACE_ISR_FUNCTIONS_INTO_FLASH

Place ISR ringbuf functions into flash

*Found in: Component config > ESP Ringbuf > CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH*

Place ISR ringbuf functions (like xRingbufferSendFromISR/xRingbufferReceiveFromISR) into flash. This frees up IRAM, but the functions can no longer be called when the cache is disabled or from an IRAM interrupt context.

This option is not compatible with ESP-IDF drivers which are configured to run the ISR from an IRAM context, e.g. CONFIG_UART_ISR_IN_IRAM.

**Default value:**

- No (disabled) if CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH
ESP System Settings Contains:

- CONFIG_ESP_SYSTEM_RTC_EXT_XTAL_BOOTSTRAP_CYCLES
- Brownout Detector
- Cache config
- CONFIG_ESP_CONSOLE_UART
- CONFIG_ESP_CONSOLE_SECONDARY
- CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ
- CONFIG_ESP_CONSOLE_USB_CDC_SUPPORT_ETS_PRINTF
- CONFIG_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP
- CONFIG_ESP_TASK_WDT_EN
- CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE
- CONFIG_ESP_SYSTEM_HW_STACK_GUARD
- CONFIG_ESP_XT_WDT
- CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL
- CONFIG_ESP_INT_WDT
- CONFIG_ESP32S2_KEEP_USB_ALIVE
- CONFIG_ESP_MAIN_TASK_AFFINITY
- CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ
- CONFIG_ESP_CONSOLE_USB_CDC_RX_BUF_SIZE
- CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE
- Trace memory
- CONFIG_ESP_CONSOLE_UART_BAUDRATE
- CONFIG_ESP_CONSOLE_UART_NUM
- CONFIG_ESP_CONSOLE_UART_RX_GPIO
- CONFIG_ESP_CONSOLE_UART_TX_GPIO
- Trace memory
- CONFIG_ESP_CONSOLE_UART_BAUDRATE
- CONFIG_ESP_CONSOLE_UART_NUM
- CONFIG_ESP_CONSOLE_UART_RX_GPIO
- CONFIG_ESP_CONSOLE_UART_TX_GPIO

CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ

CPU frequency

Found in: Component config > ESP System Settings

CPU frequency to be set on application startup.

Available options:

- 40 MHz (CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_40)
- 80 MHz (CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_80)
- 160 MHz (CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_160)
- 240 MHz (CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240)

Cache config Contains:

- CONFIG_ESP32S2_DATA_CACHE_LINE_SIZE
- CONFIG_ESP32S2_DATA_CACHE_SIZE
- CONFIG_ESP32S2_DATA_CACHE_WRAP
- CONFIG_ESP32S2_INSTRUCTION_CACHE_WRAP
- CONFIG_ESP32S2_INSTRUCTION_CACHE_LINE_SIZE
- CONFIG_ESP32S2_INSTRUCTION_CACHE_SIZE
CONFIG_ESP32S2_INSTRUCTION_CACHE_SIZE
Instruction cache size

*Found in: Component config > ESP System Settings > Cache config*

Instruction cache size to be set on application startup. If you use 8KB instruction cache rather than 16KB instruction cache, then the other 8KB will be added to the heap.

Available options:

- 8KB (CONFIG_ESP32S2_INSTRUCTION_CACHE_8KB)
- 16KB (CONFIG_ESP32S2_INSTRUCTION_CACHE_16KB)

CONFIG_ESP32S2_INSTRUCTION_CACHE_LINE_SIZE
Instruction cache line size

*Found in: Component config > ESP System Settings > Cache config*

Instruction cache line size to be set on application startup.

Available options:

- 16 Bytes (CONFIG_ESP32S2_INSTRUCTION_CACHE_LINE_16B)
- 32 Bytes (CONFIG_ESP32S2_INSTRUCTION_CACHE_LINE_32B)

CONFIG_ESP32S2_DATA_CACHE_SIZE
Data cache size

*Found in: Component config > ESP System Settings > Cache config*

Data cache size to be set on application startup. If you use 0KB data cache, the other 16KB will be added to the heap. If you use 8KB data cache rather than 16KB data cache, the other 8KB will be added to the heap.

Available options:

- 0KB (CONFIG_ESP32S2_DATA_CACHE_0KB)
- 8KB (CONFIG_ESP32S2_DATA_CACHE_8KB)
- 16KB (CONFIG_ESP32S2_DATA_CACHE_16KB)

CONFIG_ESP32S2_DATA_CACHE_LINE_SIZE
Data cache line size

*Found in: Component config > ESP System Settings > Cache config*

Data cache line size to be set on application startup.

Available options:

- 16 Bytes (CONFIG_ESP32S2_DATA_CACHE_LINE_16B)
- 32 Bytes (CONFIG_ESP32S2_DATA_CACHE_LINE_32B)
**CONFIG_ESP32S2_INSTRUCTION_CACHE_WRAP**

Enable instruction cache wrap

*Found in: Component config > ESP System Settings > Cache config*

If enabled, instruction cache will use wrap mode to read SPI flash (maybe SPI RAM). The wrap length equals to INSTRUCTION_CACHE_LINE_SIZE. However, it depends on complex conditions.

**Default value:**
- No (disabled)

**CONFIG_ESP32S2_DATA_CACHE_WRAP**

Enable data cache wrap

*Found in: Component config > ESP System Settings > Cache config*

If enabled, data cache will use wrap mode to read SPI RAM (maybe SPI flash). The wrap length equals to DATA_CACHE_LINE_SIZE. However, it depends on complex conditions.

**Default value:**
- No (disabled)

**Memory** Contains:

- `CONFIG_ESP32S2_RTCDATA_IN_FAST_MEM`
- `CONFIG_ESP32S2_USE_FIXED_STATIC_RAM_SIZE`

**CONFIG_ESP32S2_RTCDATA_IN_FAST_MEM**

Place RTC_DATA_ATTR and RTC_RODATA_ATTR variables into RTC fast memory segment

*Found in: Component config > ESP System Settings > Memory*

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.

**Default value:**
- No (disabled)

**CONFIG_ESP32S2_USE_FIXED_STATIC_RAM_SIZE**

Use fixed static RAM size

*Found in: Component config > ESP System Settings > Memory*

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 ESP32S2_FIXED_STATIC_RAM_SIZE

**Default value:**
- No (disabled)

**CONFIG_ESP32S2_FIXED_STATIC_RAM_SIZE**

Fixed Static RAM size

*Found in: Component config > ESP System Settings > Memory > CON-
FIG_ESP32S2_USE_FIXED_STATIC_RAM_SIZE*
RAM size dedicated for static variables (.data & .bss sections). This value is less than the chips total memory, as not all of it can be used for this purpose. E.g. parts are used by the software bootloader, and will only be available as heap memory after app startup.

Range:
- from 0 to 0x34000 if `CONFIG_ESP32S2_USE_FIXED_STATIC_RAM_SIZE`

Default value:
- “0x10000” if `CONFIG_ESP32S2_USE_FIXED_STATIC_RAM_SIZE`

**Trace memory**
Contains:
- `CONFIG_ESP32S2_TRAX`

**CONFIG_ESP32S2_TRAX**
Use TRAX tracing feature

*Found in: Component config > ESP System Settings > Trace memory*

The ESP32S2 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.

Default value:
- No (disabled)

**CONFIG_ESP_SYSTEM_PANIC**
Panic handler behaviour

*Found in: Component config > ESP System Settings*

If FreeRTOS detects unexpected behaviour or an unhandled exception, the panic handler is invoked. Configure the panic handler’s action here.

Available options:

- Print registers and halt (CONFIG_ESP_SYSTEM_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 (CONFIG_ESP_SYSTEM_PANIC_PRINT_REBOOT)
  Outputs the relevant registers over the serial port and immediately reset the processor.
- Silent reboot (CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT)
  Just resets the processor without outputting anything
- GDBStub on panic (CONFIG_ESP_SYSTEM_PANIC_GDBSTUB)
  Invoke gdbstub on the serial port, allowing for gdb to attach to it to do a postmortem of the crash.

**CONFIG_ESP_SYSTEM_PANIC_REBOOT_DELAY SECONDS**
Panic reboot delay (Seconds)

*Found in: Component config > ESP System Settings*

After the panic handler executes, you can specify a number of seconds to wait before the device reboots.

Range:
- from 0 to 99

Default value:
- 0
CONFIG_ESP_SYSTEM_RTC_EXT_XTAL_BOOTSTRAP_CYCLES

Bootstrap cycles for external 32kHz crystal

*Found in: Component config > ESP System Settings*

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_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP

Enable RTC fast memory for dynamic allocations

*Found in: Component config > ESP System Settings*

This config option allows to add RTC fast memory region to system heap with capability similar to that of DRAM region but without DMA. This memory will be consumed first per heap initialization order by early startup services and scheduler related code. Speed wise RTC fast memory operates on APB clock and hence does not have much performance impact.

**Default value:**
- Yes (enabled)

**Memory protection** Contains:
- CONFIG_ESP_SYSTEM_PMP_IDRAM_SPLIT
- CONFIG_ESP_SYSTEM_MEMPROT_FEATURE

CONFIG_ESP_SYSTEM_PMP_IDRAM_SPLIT

Enable IRAM/DRAM split protection

*Found in: Component config > ESP System Settings > Memory protection*

If enabled, the CPU watches all the memory access and raises an exception in case of any memory violation. This feature automatically splits the SRAM memory, using PMP, into data and instruction segments and sets Read/Execute permissions for the instruction part (below given splitting address) and Read/Write permissions for the data part (above the splitting address). The memory protection is effective on all access through the IRAM0 and DRAM0 buses.

**Default value:**
- Yes (enabled) if SOC_CPU_IDRAM_SPLIT_USING_PMP

CONFIG_ESP_SYSTEM_MEMPROT_FEATURE

Enable memory protection

*Found in: Component config > ESP System Settings > Memory protection*

If enabled, the permission control module watches all the memory access and fires the panic handler if a permission violation is detected. This feature automatically splits the SRAM memory into data and instruction segments and sets Read/Execute permissions for the instruction part (below given splitting address) and Read/Write permissions for the data part (above the splitting address). The memory protection is effective on all access through the IRAM0 and DRAM0 buses.

**Default value:**
- Yes (enabled)
Chapter 2. API Reference

CONFIG_ESP_SYSTEM_MEMPROT_FEATURE_LOCK
Lock memory protection settings

*Found in: Component config > ESP System Settings > Memory protection > CONFIG_ESP_SYSTEM_MEMPROT_FEATURE*

Once locked, memory protection settings cannot be changed anymore. The lock is reset only on the chip startup.

**Default value:**
- Yes (enabled)

CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE
System event queue size

*Found in: Component config > ESP System Settings*

Config system event queue size in different application.

**Default value:**
- 32

CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE
Event loop task stack size

*Found in: Component config > ESP System Settings*

Config system event task stack size in different application.

**Default value:**
- 2304

CONFIG_ESP_MAIN_TASK_STACK_SIZE
Main task stack size

*Found in: Component config > ESP System Settings*

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.

**Default value:**
- 3584

CONFIG_ESP_MAIN_TASK_AFFINITY
Main task core affinity

*Found in: Component config > ESP System Settings*

Configure the “main task” core affinity. This is the used core of the task which calls app_main(). If app_main() returns then this task is deleted.

**Available options:**
- CPU0 (CONFIG_ESP_MAIN_TASK_AFFINITY_CPU0)
- CPU1 (CONFIG_ESP_MAIN_TASK_AFFINITY_CPU1)
- No affinity (CONFIG_ESP_MAIN_TASK_AFFINITY_NO_AFFINITY)
CONFIG_ESP_MINIMAL_SHARED_STACK_SIZE

Minimal allowed size for shared stack

*Found in:* Component config > ESP System Settings

Minimal value of size, in bytes, accepted to execute an expression with shared stack.

**Default value:**
- 2048

CONFIG_ESP_CONSOLE_UART

Channel for console output

*Found in:* Component config > ESP System Settings

Select where to send console output (through stdout and stderr).

- Default is to use UART0 on pre-defined GPIOs.
- 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 ROM output can be suppressed by GPIO strapping or EFUSE, refer to chip datasheet for details.
- On chips with USB OTG peripheral, “USB CDC” option redirects output to the CDC port. This option uses the CDC driver in the chip ROM. This option is incompatible with TinyUSB stack.
- On chips with an USB serial/JTAG debug controller, selecting the option for that redirects output to the CDC/ACM (serial port emulation) component of that device.

**Available options:**
- Default: UART0 (CONFIG_ESP_CONSOLE_UART_DEFAULT)
- USB CDC (CONFIG_ESP_CONSOLE_USB_CDC)
- USB Serial/JTAG Controller (CONFIG_ESP_CONSOLE_USB_SERIAL_JTAG)
- Custom UART (CONFIG_ESP_CONSOLE_UART_CUSTOM)
- None (CONFIG_ESP_CONSOLE_NONE)

CONFIG_ESP_CONSOLE_SECONDARY

Channel for console secondary output

*Found in:* Component config > ESP System Settings

This secondary option supports output through other specific port like USB_SERIAL_JTAG when UART0 port as a primary is selected but not connected. This secondary output currently only supports non-blocking mode without using REPL. If you want to output in blocking mode with REPL or input through this secondary port, please change the primary config to this port in Channel for console output menu.

**Available options:**
- No secondary console (CONFIG_ESP_CONSOLE_SECONDARY_NONE)
- USB_SERIAL_JTAG_PORT (CONFIG_ESP_CONSOLE_SECONDARY_USB_SERIAL_JTAG)
  This option supports output through USB_SERIAL_JTAG port when the UART0 port is not connected. The output currently only supports non-blocking mode without using the console. If you want to output in blocking mode with REPL or input through USB_SERIAL_JTAG port, please change the primary config to ESP_CONSOLE_USB_SERIAL_JTAG above.
CONFIG_ESP_CONSOLE_UART_NUM

UART peripheral to use for console output (0-1)

*Found in: Component config > ESP System Settings*

This UART peripheral is used for console output from the ESP-IDF Bootloader and the app.

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

Due to an ESP32 ROM bug, UART2 is not supported for console output via esp_rom_printf.

Available options:

- UART0 (CONFIG_ESP_CONSOLE_UART Custom_NUM_0)
- UART1 (CONFIG_ESP_CONSOLE_UART Custom_NUM_1)

CONFIG_ESP_CONSOLE_UART_TX_GPIO

UART TX on GPIO#

*Found in: Component config > ESP System Settings*

This GPIO is used for console UART TX output in the ESP-IDF Bootloader and the app (including boot log output and default standard output and standard error of the app).

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

*Range:*
- from 0 to 45 if CONFIG_ESP_CONSOLE_UART_CUSTOM

*Default value:*
- 43 if CONFIG_ESP_CONSOLE_UART_CUSTOM

CONFIG_ESP_CONSOLE_UART_RX_GPIO

UART RX on GPIO#

*Found in: Component config > ESP System Settings*

This GPIO is used for UART RX input in the ESP-IDF Bootloader and the app (including default default standard input of the app).

Note: The default ESP-IDF Bootloader configures this pin but doesn’t read anything from the UART.

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

*Range:*
- from 0 to 46 if CONFIG_ESP_CONSOLE_UART_CUSTOM

*Default value:*
- 44 if CONFIG_ESP_CONSOLE_UART_CUSTOM

CONFIG_ESP_CONSOLE_UART_BAUDRATE

UART console baud rate

*Found in: Component config > ESP System Settings*

This baud rate is used by both the ESP-IDF Bootloader and the app (including boot log output and default standard input/output/error of the app).

The app’s maximum baud rate depends on the UART clock source. If Power Management is disabled, the UART clock source is the APB clock and all baud rates in the available range will be sufficiently
accurate. If Power Management is enabled, REF_Tick clock source is used so the baud rate is divided from 1MHz. Baud rates above 1Mbps are not possible and values between 500Kbps and 1Mbps may not be accurate.

If the configuration is different in the Bootloader binary compared to the app binary, UART is reconfigured after the bootloader exits and the app starts.

**Range:**
- from 1200 to 1000000 if `CONFIG_PM_ENABLE`

**Default value:**
- 115200

**CONFIG_ESP_CONSOLE_USB_CDC_RX_BUF_SIZE**

Size of USB CDC RX buffer

*Found in: Component config > ESP System Settings*

Set the size of USB CDC RX buffer. Increase the buffer size if your application is often receiving data over USB CDC.

**Range:**
- from 4 to 16384 if `CONFIG_ESP_CONSOLE_USB_CDC`

**Default value:**
- 64 if `CONFIG_ESP_CONSOLE_USB_CDC`

**CONFIG_ESP_CONSOLE_USB_CDC_SUPPORT_ETS_PRINTF**

Enable esp_rom_printf / ESP_EARLY_LOG via USB CDC

*Found in: Component config > ESP System Settings*

If enabled, esp_rom_printf and ESP_EARLY_LOG output will also be sent over USB CDC. Disabling this option saves about 1kB or RAM.

**Default value:**
- No (disabled) if `CONFIG_ESP_CONSOLE_USB_CDC`

**CONFIG_ESP_INT_WDT**

Interrupt watchdog

*Found in: Component config > ESP System Settings*

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.

**Default value:**
- Yes (enabled)

**CONFIG_ESP_INT_WDT_TIMEOUT_MS**

Interrupt watchdog timeout (ms)

*Found in: Component config > ESP System Settings > CONFIG_ESP_INT_WDT*

The timeout of the watchdog, in miliseconds. Make this higher than the FreeRTOS tick rate.

**Range:**
- from 10 to 10000

**Default value:**
- 300
CONFIG_ESP_INT_WDT_CHECK_CPU1

Also watch CPU1 tick interrupt

*Found in: Component config > ESP System Settings > CONFIG_ESP_INT_WDT*

Also detect if interrupts on CPU 1 are disabled for too long.

CONFIG_ESP_TASK_WDT_EN

Enable Task Watchdog Timer

*Found in: Component config > ESP System Settings*

The Task Watchdog Timer can be used to make sure individual tasks are still running. Enabling this option will enable the Task Watchdog Timer. It can be either initialized automatically at startup or initialized after startup (see Task Watchdog Timer API Reference)

**Default value:**

- Yes (enabled)

CONFIG_ESP_TASK_WDT_INIT

Initialize Task Watchdog Timer on startup

*Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN*

Enabling this option will cause the Task Watchdog Timer to be initialized automatically at startup.

**Default value:**

- Yes (enabled)

CONFIG_ESP_TASK_WDT_PANIC

Invoke panic handler on Task Watchdog timeout

*Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT*

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)

**Default value:**

- No (disabled)

CONFIG_ESP_TASK_WDT_TIMEOUT_S

Task Watchdog timeout period (seconds)

*Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT*

Timeout period configuration for the Task Watchdog Timer in seconds. This is also configurable at run time (see Task Watchdog Timer API Reference)

**Range:**

- from 1 to 60

**Default value:**

- 5
CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0
Watch CPU0 Idle Task

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT

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.

Default value:
• Yes (enabled)

CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1
Watch CPU1 Idle Task

Found in: Component config > ESP System Settings > CONFIG_ESP_TASK_WDT_EN > CONFIG_ESP_TASK_WDT_INIT

If this option is enabled, the Task Watchdog Timer will watch the CPU1 Idle Task.

CONFIG_ESP_XT_WDT
Initialize XTAL32K watchdog timer on startup

Found in: Component config > ESP System Settings

This watchdog timer can detect oscillation failure of the XTAL32K_CLK. When such a failure is detected the hardware can be set up to automatically switch to BACKUP32K_CLK and generate an interrupt.

CONFIG_ESP_XT_WDT_TIMEOUT
XTAL32K watchdog timeout period

Found in: Component config > ESP System Settings > CONFIG_ESP_XT_WDT

Timeout period configuration for the XTAL32K watchdog timer based on RTC_CLK.
Range:
• from 1 to 255 if CONFIG_ESP_XT_WDT

Default value:
• 200 if CONFIG_ESP_XT_WDT

CONFIG_ESP_XT_WDT_BACKUP_CLK_ENABLE
Automatically switch to BACKUP32K_CLK when timer expires

Found in: Component config > ESP System Settings > CONFIG_ESP_XT_WDT

Enable this to automatically switch to BACKUP32K_CLK as the source of RTC_SLOW_CLK when the watchdog timer expires.

Default value:
• Yes (enabled) if CONFIG_ESP_XT_WDT
**CONFIG_ESP_PANIC_HANDLER_IRAM**

Place panic handler code in IRAM

*Found in: Component config > ESP System Settings*

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 (including required UART functions) is placed in IRAM. This may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash) or when flash cache is corrupted when an exception is triggered.

**Default value:**
- No (disabled)

**CONFIG_ESP_DEBUG_STUBS_ENABLE**

OpenOCD debug stubs

*Found in: Component config > ESP System Settings*

Debug stubs are used by OpenOCD to execute pre-compiled onboard code which does some useful debugging stuff, e.g. GCOV data dump.

**CONFIG_ESP_DEBUG_OCD_AWARE**

Make exception and panic handlers JTAG/OCD aware

*Found in: Component config > ESP System Settings*

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.

**Default value:**
- Yes (enabled)

**CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL**

Interrupt level to use for Interrupt Watchdog and other system checks

*Found in: Component config > ESP System Settings*

Interrupt level to use for Interrupt Watchdog, IPC_ISR and other system checks.

Available options:

- Level 5 interrupt (CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL_5)
  Using level 5 interrupt for Interrupt Watchdog, IPC_ISR and other system checks.
- Level 4 interrupt (CONFIG_ESP_SYSTEM_CHECK_INT_LEVEL_4)
  Using level 4 interrupt for Interrupt Watchdog, IPC_ISR and other system checks.

**Brownout Detector** Contains:

- CONFIG_ESP_BROWNOUT_DET
CONFIG_ESP_BROWNOUT_DET

Hardware brownout detect & reset

*Found in: Component config > ESP System Settings > Brownout Detector*

The ESP32-S2 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.

**Default value:**
- Yes (enabled)

CONFIG_ESP_BROWNOUT_DET_LVL_SEL

Brownout voltage level

*Found in: Component config > ESP System Settings > Brownout Detector > CONFIG_ESP_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-S2 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:**
- 2.44V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_7)
- 2.56V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_6)
- 2.67V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_5)
- 2.84V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_4)
- 2.98V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_3)
- 3.19V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_2)
- 3.30V (CONFIG_ESP_BROWNOUT_DET_LVL_SEL_1)

CONFIG_ESP32S2_KEEP_USB_ALIVE

Keep USB peripheral enabled at startup

*Found in: Component config > ESP System Settings*

During the application initialization process, all the peripherals except UARTs and timers are reset. Enable this option to keep USB peripheral enabled. This option is automatically enabled if USB CDC console is selected.

**Default value:**
- Yes (enabled) if CONFIG_ESP_CONSOLE_USB_CDC

CONFIG_ESP_SYSTEM_HW_STACK_GUARD

Hardware stack guard

*Found in: Component config > ESP System Settings*

This config allows to trigger a panic interrupt when Stack Pointer register goes out of allocated stack memory bounds.

**Default value:**
- Yes (enabled) if SOC_ASSIST_DEBUG_SUPPORTED
IPC (Inter-Processor Call)  Contains:

- `CONFIG_ESP_IPC_TASK_STACK_SIZE`
- `CONFIG_ESP_IPC_USES_CALLERS_PRIORITY`

**CONFIG_ESP_IPC_TASK_STACK_SIZE**

Inter-Processor Call (IPC) task stack size

*Found in: Component config > IPC (Inter-Processor Call)*

Configure the IPC tasks stack size. An 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 IPC stack size should be enough for most common simple use cases. However, users can increase/decrease the stack size to their needs.

**Range:**
- from 512 to 65536

**Default value:**
- 1024

**CONFIG_ESP_IPC_USES_CALLERS_PRIORITY**

IPC runs at caller’s priority

*Found in: Component config > IPC (Inter-Processor Call)*

If this option is not enabled then the IPC task will keep behavior same as prior to that of ESP-IDF v4.0, hence IPC task will run at (configMAX_PRIORITIES - 1) priority.

**High resolution timer (esp_timer)**  Contains:

- `CONFIG_ESP_TIMER_PROFILING`
- `CONFIG_ESP_TIMER_TASK_AFFINITY`
- `CONFIG_ESP_TIMER_TASK_STACK_SIZE`
- `CONFIG_ESP_TIMER_INTERRUPT_LEVEL`
- `CONFIG_ESP_TIMER_SHOW_EXPERIMENTAL`
- `CONFIG_ESP_TIMER_SUPPORTS_ISR_DISPATCH_METHOD`
- `CONFIG_ESP_TIMER_ISR_AFFINITY`

**CONFIG_ESP_TIMER_PROFILING**

Enable esp_timer profiling features

*Found in: Component config > High resolution timer (esp_timer)*

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.

**Default value:**
- No (disabled)

**CONFIG_ESP_TIMER_TASK_STACK_SIZE**

High-resolution timer task stack size

*Found in: Component config > High resolution timer (esp_timer)*

Configure the stack size of “timer_task” 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”.

**Range:**
- from 2048 to 65536

**Default value:**
- 3584

**CONFIG_ESP_TIMER_INTERRUPT_LEVEL**

Interrupt level

*Found in: Component config > High resolution timer (esp_timer)*

It sets the interrupt level for esp_timer ISR in range 1..3. A higher level (3) helps to decrease the ISR esp_timer latency.

**Range:**
- from 1 to 1

**Default value:**
- 1

**CONFIG_ESP_TIMER_SHOW_EXPERIMENTAL**

show esp_timer’s experimental features

*Found in: Component config > High resolution timer (esp_timer)*

This shows some hidden features of esp_timer. Note that they may break other features, use them with care.

**CONFIG_ESP_TIMER_TASK_AFFINITY**

esp_timer task core affinity

*Found in: Component config > High resolution timer (esp_timer)*

The default settings: timer TASK on CPU0 and timer ISR on CPU0. Other settings may help in certain cases, but note that they may break other features, use them with care.

- “CPU0” (default) esp_timer task is processed by CPU0.
- “CPU1” : esp_timer task is processed by CPU1.
- “No affinity” : esp_timer task can be processed by any CPU.

**Available options:**

- CPU0 (CONFIG_ESP_TIMER_TASK_AFFINITY_CPU0)
- CPU1 (CONFIG_ESP_TIMER_TASK_AFFINITY_CPU1)
- No affinity (CONFIG_ESP_TIMER_TASK_AFFINITY_NO_AFFINITY)

**CONFIG_ESP_TIMER_ISR_AFFINITY**

timer interrupt core affinity

*Found in: Component config > High resolution timer (esp_timer)*

The default settings: timer TASK on CPU0 and timer ISR on CPU0. Other settings may help in certain cases, but note that they may break other features, use them with care.

- “CPU0” : (default) timer interrupt is processed by CPU0.
- “CPU1” : timer interrupt is processed by CPU1.
- “No affinity” : timer interrupt can be processed by any CPU. It helps to reduce latency but there is a disadvantage it leads to the timer ISR running on every core. It increases the CPU time usage for timer ISRs by N on an N-core system.

**Available options:**

- CPU0 (CONFIG_ESP_TIMER_ISR_AFFINITY_CPU0)
- CPU1 (CONFIG_ESP_TIMER_ISR_AFFINITY_CPU1)
- No affinity (CONFIG_ESP_TIMER_ISR_AFFINITY_NO_AFFINITY)
Chapter 2. API Reference

- CPU0 (CONFIG_ESP_TIMER_ISR_AFFINITY_CPU0)
- CPU1 (CONFIG_ESP_TIMER_ISR_AFFINITY_CPU1)
- No affinity (CONFIG_ESP_TIMER_ISR_AFFINITY_NO_AFFINITY)

**CONFIG_ESP_TIMER_SUPPORTS_ISR_DISPATCH_METHOD**

Support ISR dispatch method

*Found in:* Component config > High resolution timer (esp_timer)

Allows using ESP_TIMER_ISR dispatch method (ESP_TIMER_TASK dispatch method is also available). - ESP_TIMER_TASK - Timer callbacks are dispatched from a high-priority esp_timer task. - ESP_TIMER_ISR - Timer callbacks are dispatched directly from the timer interrupt handler. The ISR dispatch can be used, in some cases, when a callback is very simple or need a lower-latency.

**Default value:**
- No (disabled)

**Wi-Fi** Contains:

- CONFIG_ESP_WIFI_TESTING_OPTIONS
- CONFIG_ESP_WIFI_WPS_SOFTAP_REGISTRAR
- CONFIG_ESP_WIFI_11KV_SUPPORT
- CONFIG_ESP_WIFI_11R_SUPPORT
- CONFIG_ESP_WIFI_DPP_SUPPORT
- CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT
- CONFIG_ESP_WIFI_MBO_SUPPORT
- CONFIG_ESP_WIFI_SUITE_B_192
- CONFIG_ESP_WIFI_ENABLE_WPA3_OWE_STA
- CONFIG_ESP_WIFI_WAPI_PSK
- CONFIG_ESP_WIFI_ENABLE_WIFI_RX_STATS
- CONFIG_ESP_WIFI_ENABLE_WIFI_TX_STATS
- CONFIG_ESP_WIFI_ENABLE_WPA3_SAE
- CONFIG_ESP_WIFI_SOFTAP_BEACON_MAX_LEN
- CONFIG_ESP_WIFI_CACHE_TX_BUFFER_NUM
- CONFIG_ESP_WIFI_DYNAMIC_RX_BUFFER_NUM
- CONFIG_ESP_WIFI_DYNAMIC_TX_BUFFER_NUM
- CONFIG_ESP_WIFI_RX_MGMT_BUF_NUM_DEF
- CONFIG_ESP_WIFI_STATIC_RX_BUFFER_NUM
- CONFIG_ESP_WIFI_STATIC_TX_BUFFER_NUM
- CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM
- CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE
- CONFIG_ESP_WIFI_DEBUG_PRINT
- CONFIG_ESP_WIFI_MGMT_RX_BUFFER
- CONFIG_ESP_WIFI_MGMT_TX_BUFFER
- CONFIG_ESP_WIFI_MBEDTLS_CRYPTO
- CONFIG_ESP_WIFI_AMSDU_TX_ENABLED
- CONFIG_ESP_WIFI_NAN_ENABLE
- CONFIG_ESP_WIFI_CSI_ENABLED
- CONFIG_ESP_WIFI_EXTRA伊拉M_OPT
- CONFIG_ESP_WIFI_FTM_ENABLE
- CONFIG_ESP_WIFI_GCMP_SUPPORT
- CONFIG_ESP_WIFI_GMAC_SUPPORT
- CONFIG_ESP_WIFI_IRAM_OPT
- CONFIG_ESP_WIFI_MGMT_SBUF_NUM
- CONFIG_ESP_WIFI_ENHANCED_LIGHT_SLEEP
### CONFIG_ESP_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 ESP_WIFI_AMPDU_RX_ENABLED is enabled, this value is recommended to set equal or bigger than ESP_WIFI_RX_BA_WIN in order to achieve better throughput and compatibility with both stations and APs.

*Range:*  
- from 2 to 128 if SOC_WIFI_HE_SUPPORT

*Default value:*  
- 16 if CONFIG_SPIRAM.Try_ALLOCATE_WIFI_LWIP

### CONFIG_ESP_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.

*Range:*  
- from 0 to 1024 if CONFIG_LWIP.WND_SCALE

*Default value:*  
- 32

### CONFIG_ESP_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 (CONFIG_ESP_WIFI_STATIC_TX_BUFFER)
- Dynamic (CONFIG_ESP_WIFI_DYNAMIC_TX_BUFFER)

**CONFIG_ESP_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.

*Range:*
  - from 1 to 64 if **CONFIG_ESP_WIFI_STATIC_TX_BUFFER**

*Default value:*
  - 16 if **CONFIG_ESP_WIFI_STATIC_TX_BUFFER**

**CONFIG_ESP_WIFI_CACHE_TX_BUFFER_NUM**

Max number of WiFi cache TX buffers

*Found in: Component config > Wi-Fi*

Set the number of WiFi cache TX buffer number.

For each TX packet from uplayer, such as LWIP etc, WiFi driver needs to allocate a static TX buffer and makes a copy of uplayer packet. If WiFi driver fails to allocate the static TX buffer, it caches the uplayer packets to a dedicated buffer queue, this option is used to configure the size of the cached TX queue.

*Range:*
  - from 16 to 128 if **CONFIG_SPIRAM**

*Default value:*
  - 32 if **CONFIG_SPIRAM**

**CONFIG_ESP_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.

*Range:*
  - from 1 to 128
Chapter 2. API Reference

**Default value:**
- 32

**CONFIG_ESP_WIFI_MGMT_RX_BUFFER**
Type of WiFi RX MGMT buffers

*Found in: Component config > Wi-Fi*

Select type of WiFi RX MGMT buffers:

If “Static” is selected, WiFi RX MGMT buffers are allocated when WiFi is initialized and released when WiFi is de-initialized. The size of each static RX MGMT buffer is fixed to about 500 Bytes.

If “Dynamic” is selected, each WiFi RX MGMT buffer is allocated as needed when a MGMT data frame is received. The MGMT buffer is freed after the MGMT data frame has been processed by the WiFi driver.

Available options:

- Static (CONFIG_ESP_WIFI_STATIC_RX_MGMT_BUFFER)
- Dynamic (CONFIG_ESP_WIFI_DYNAMIC_RX_MGMT_BUFFER)

**CONFIG_ESP_WIFI_RX_MGMT_BUF_NUM_DEF**
Max number of WiFi RX MGMT buffers

*Found in: Component config > Wi-Fi*

Set the number of WiFi RX_MGMT buffers.

For Management buffers, the number of dynamic and static management buffers is the same. In order to prevent memory fragmentation, the management buffer type should be set to static first.

**Range:**
- from 1 to 10

**Default value:**
- 5

**CONFIG_ESP_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_ESP_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.

**Default value:**
- No (disabled)

**CONFIG_ESP_WIFI_AMPDU_TX_ENABLED**
WiFi AMPDU TX

*Found in: Component config > Wi-Fi*

Select this option to enable AMPDU TX feature

**Default value:**
- Yes (enabled)
**CONFIG_ESP_WIFI_TX_BA_WIN**

WiFi AMPDU TX BA window size

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_TX_BA_WIN*

Set the size of WiFi Block Ack TX window. Generally a bigger value means higher throughput but more memory. Most of the 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.

*Range:*
  - from 2 to 64 if SOC_WIFI_HE_SUPPORT && CONFIG_ESP_WIFI_AMPDU_TX_ENABLED

*Default value:*
  - 6

**CONFIG_ESP_WIFI_AMPDU_RX_ENABLED**

WiFi AMPDU RX

*Found in: Component config > Wi-Fi*

Select this option to enable AMPDU RX feature

*Default value:*
  - Yes (enabled)

**CONFIG_ESP_WIFI_RX_BA_WIN**

WiFi AMPDU RX BA window size

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_RX_BA_WIN*

Set the size of WiFi Block Ack RX window. Generally a bigger value means higher throughput and better compatibility but more memory. Most of the 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 prefered to alloc in PSRAM first, the default and minimum value should be 16 to achieve better throughput and compatibility with both stations and APs.

*Range:*
  - from 2 to 64 if SOC_WIFI_HE_SUPPORT && CONFIG_ESP_WIFI_AMPDU_RX_ENABLED

*Default value:*
  - 16 if CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP && CONFIG_ESP_WIFI_AMPDU_RX_ENABLED

**CONFIG_ESP_WIFI_AMSDU_TX_ENABLED**

WiFi AMSDU TX

*Found in: Component config > Wi-Fi*

Select this option to enable AMSDU TX feature

*Default value:*
  - No (disabled) if CONFIG_SPIRAM

**CONFIG_ESP_WIFI_NVS_ENABLED**
WiFi NVS flash

*Found in: Component config > Wi-Fi*

Select this option to enable WiFi NVS flash

**Default value:**
- Yes (enabled)

**CONFIG_ESP_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 (CONFIG_ESP_WIFI_TASK_PINNED_TO_CORE_0)
- Core 1 (CONFIG_ESP_WIFI_TASK_PINNED_TO_CORE_1)

**CONFIG_ESP_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.

**Range:**
- from 752 to 1256

**Default value:**
- 752

**CONFIG_ESP_WIFI_MGMT_SBUF_NUM**

WiFi mgmt short buffer number

*Found in: Component config > Wi-Fi*

Set the number of WiFi management short buffer.

**Range:**
- from 6 to 32

**Default value:**
- 32
CONFIG_ESP_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.

**Default value:**
- Yes (enabled)

CONFIG_ESP_WIFI_EXTRA_IRAM_OPT

WiFi EXTRA IRAM speed optimization

*Found in: Component config > Wi-Fi*

Select this option to place additional frequently called Wi-Fi library functions in IRAM. When this option is disabled, more than 5Kbytes of IRAM memory will be saved but Wi-Fi throughput will be reduced.

**Default value:**
- No (disabled)

CONFIG_ESP_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.

**Default value:**
- Yes (enabled)

CONFIG_ESP_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.

**Default value:**
- Yes (enabled)

CONFIG_ESP_WIFI_ENABLE_SAE_PK

Enable SAE-PK

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_ENABLE_WPA3_SAE*

Select this option to enable SAE-PK

**Default value:**
- Yes (enabled)
**CONFIG_ESP_WIFI_SOFTAP_SAE_SUPPORT**

Enable WPA3 Personal(SAE) SoftAP

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_ENABLE_WPA3_SAE*

Select this option to enable SAE support in softAP mode.

**Default value:**
- Yes (enabled)

**CONFIG_ESP_WIFI_ENABLE_WPA3_OWE_STA**

Enable OWE STA

*Found in: Component config > Wi-Fi*

Select this option to allow the device to establish OWE 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.

**Default value:**
- Yes (enabled)

**CONFIG_ESP_WIFI_SLP_IRAM_OPT**

WiFi SLP IRAM speed optimization

*Found in: Component config > Wi-Fi*

Select this option to place called Wi-Fi library TBTT process and receive beacon functions in IRAM. Some functions can be put in IRAM either by ESP_WIFI_IRAM_OPT and ESP_WIFI_RX_IRAM_OPT, or this one. If already enabled ESP_WIFI_IRAM_OPT, the other 7.3KB IRAM memory would be taken by this option. If already enabled ESP_WIFI_RX_IRAM_OPT, the other 1.3KB IRAM memory would be taken by this option. If neither of them are enabled, the other 7.4KB IRAM memory would be taken by this option. Wi-Fi power-save mode average current would be reduced if this option is enabled.

**CONFIG_ESP_WIFI_SLP_DEFAULT_MIN_ACTIVE_TIME**

Minimum active time

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_IRAM_OPT*

The minimum timeout for waiting to receive data, unit: milliseconds.

**Range:**
- from 8 to 60 if CONFIG_ESP_WIFI_SLP_IRAM_OPT

**Default value:**
- 50 if CONFIG_ESP_WIFI_SLP_IRAM_OPT

**CONFIG_ESP_WIFI_SLP_DEFAULT_MAX_ACTIVE_TIME**

Maximum keep alive time

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_IRAM_OPT*

The maximum time that wifi keep alive, unit: seconds.

**Range:**
- from 10 to 60 if CONFIG_ESP_WIFI_SLP_IRAM_OPT

**Default value:**
- 10 if CONFIG_ESP_WIFI_SLP_IRAM_OPT
CONFIG_ESP_WIFI_FTM_ENABLE

WiFi FTM

*Found in: Component config > Wi-Fi*

Enable feature Fine Timing Measurement for calculating WiFi Round-Trip-Time (RTT).

**Default value:**
- No (disabled)

CONFIG_ESP_WIFI_FTM_INITIATOR_SUPPORT

FTM Initiator support

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_FTM_ENABLE*

**Default value:**
- Yes (enabled) if CONFIG_ESP_WIFI_FTM_ENABLE

CONFIG_ESP_WIFI_FTM_RESPONDER_SUPPORT

FTM Responder support

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_FTM_ENABLE*

**Default value:**
- Yes (enabled) if CONFIG_ESP_WIFI_FTM_ENABLE

CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE

Power Management for station at disconnected

*Found in: Component config > Wi-Fi*

Select this option to enable power_management for station when disconnected. Chip will do modem-sleep when rf module is not in use any more.

**Default value:**
- Yes (enabled)

CONFIG_ESP_WIFI_GCMP_SUPPORT

WiFi GCMP Support(GCMP128 and GCMP256)

*Found in: Component config > Wi-Fi*

Select this option to enable GCMP support. GCMP support is compulsory for WiFi Suite-B support.

**Default value:**
- No (disabled) if SOC_WIFI_GCMP_SUPPORT

CONFIG_ESP_WIFI_GMAC_SUPPORT

WiFi GMAC Support(GMAC128 and GMAC256)

*Found in: Component config > Wi-Fi*

Select this option to enable GMAC support. GMAC support is compulsory for WiFi 192 bit certification.

**Default value:**
- No (disabled)
CONFIG_ESP_WIFI_SOFTAP_SUPPORT
WiFi SoftAP Support

*Found in: Component config > Wi-Fi*

WiFi module can be compiled without SoftAP to save code size.

**Default value:**
- Yes (enabled)

CONFIG_ESP_WIFI_ENHANCED_LIGHT_SLEEP
WiFi modem automatically receives the beacon

*Found in: Component config > Wi-Fi*

The wifi modem automatically receives the beacon frame during light sleep.

**Default value:**
- No (disabled) if CONFIG_ESP_PHY_MAC_BB_PD && SOC_PM_SUPPORT_BEACON_WAKEUP

CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT
Wifi sleep optimize when beacon lost

*Found in: Component config > Wi-Fi*

Enable wifi sleep optimization when beacon loss occurs and immediately enter sleep mode when the WiFi module detects beacon loss.

CONFIG_ESP_WIFI_SLP_BEACON_LOST_TIMEOUT
Beacon loss timeout

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Timeout time for close rf phy when beacon loss occurs, Unit: 1024 microsecond.

**Range:**
- from 5 to 100 if CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

**Default value:**
- 10 if CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

CONFIG_ESP_WIFI_SLP_BEACON_LOST_THRESHOLD
Maximum number of consecutive lost beacons allowed

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Maximum number of consecutive lost beacons allowed, WiFi keeps Rx state when the number of consecutive beacons lost is greater than the given threshold.

**Range:**
- from 0 to 8 if CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT

**Default value:**
- 3 if CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT
CONFIG_ESP_WIFI_SLP_PHY_ON_DELTA_EARLY_TIME

Delta early time for RF PHY on

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Delta early time for rf phy on, When the beacon is lost, the next rf phy on will be earlier the time specified by the configuration item, Unit: 32 microsecond.

**Range:**
- from 0 to 100 if `CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT`

**Default value:**
- 2 if `CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT`

CONFIG_ESP_WIFI_SLP_PHY_OFF_DELTA_TIMEOUT_TIME

Delta timeout time for RF PHY off

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT*

Delta timeout time for rf phy off, When the beacon is lost, the next rf phy off will be delayed for the time specified by the configuration item. Unit: 1024 microsecond.

**Range:**
- from 0 to 8 if `CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT`

**Default value:**
- 2 if `CONFIG_ESP_WIFI_SLP_BEACON_LOST_OPT`

CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM

Maximum espnow encrypt peers number

*Found in: Component config > Wi-Fi*

Maximum number of encrypted peers supported by espnow. The number of hardware keys for encryption is fixed. And the espnow and SoftAP share the same hardware keys. So this configuration will affect the maximum connection number of SoftAP. Maximum espnow encrypted peers number + maximum number of connections of SoftAP = Max hardware keys number. When using ESP mesh, this value should be set to a maximum of 6.

**Range:**
- from 0 to 17

**Default value:**
- 7

CONFIG_ESP_WIFI_NAN_ENABLE

WiFi Aware

*Found in: Component config > Wi-Fi*

Enable WiFi Aware (NAN) feature.

**Default value:**
- No (disabled)

CONFIG_ESP_WIFI_ENABLE_WIFI_TX_STATS

Enable Wi-Fi transmission statistics

*Found in: Component config > Wi-Fi*

Enable Wi-Fi transmission statistics. Total support 4 access category. Each access category will use 346 bytes memory.
**Default value:**
- Yes (enabled) if SOC_WIFI_HE_SUPPORT

**CONFIG_ESP_WIFI_MBEDTLS_CRYPTO**

Use MbedTLS crypto APIs

*Found in: Component config > Wi-Fi*

Select this option to enable the use of MbedTLS crypto APIs. The internal crypto support within the supplicant is limited and may not suffice for all new security features, including WPA3.

It is recommended to always keep this option enabled. Additionally, note that MbedTLS can leverage hardware acceleration if available, resulting in significantly faster cryptographic operations.

**Default value:**
- Yes (enabled)

**CONFIG_ESP_WIFI_MBEDTLS_TLS_CLIENT**

Use MbedTLS TLS client for WiFi Enterprise connection

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_MBEDTLS_CRYPTO*

Select this option to use MbedTLS TLS client for WPA2 enterprise connection. Please note that from MbedTLS-3.0 onwards, MbedTLS does not support SSL-3.0, TLS-v1.0, TLS-v1.1 versions. In case your server is using one of these versions, it is advisable to update your server. Please disable this option for compatibility with older TLS versions.

**Default value:**
- Yes (enabled)

**CONFIG_ESP_WIFI_WAPI_PSK**

Enable WAPI PSK support

*Found in: Component config > Wi-Fi*

Select this option to enable WAPI-PSK which is a Chinese National Standard Encryption for Wireless LANs (GB 15629.11-2003).

**Default value:**
- No (disabled)

**CONFIG_ESP_WIFI_SUITE_B_192**

Enable NSA suite B support with 192 bit key

*Found in: Component config > Wi-Fi*

Select this option to enable 192 bit NSA suite-B. This is necessary to support WPA3 192 bit security.

**Default value:**
- No (disabled) if SOC_WIFI_GCMP_SUPPORT

**CONFIG_ESP_WIFI_11KV_SUPPORT**

Enable 802.11k, 802.11v APIs Support

*Found in: Component config > Wi-Fi*

Select this option to enable 802.11k 802.11v APIs (RRM and BTM support). Only APIs which are helpful for network assisted roaming are supported for now. Enable this option with BTM and RRM enabled in sta config to make device ready for network assisted roaming. BTM: BSS transition management enables an AP to request a station to transition to a specific AP, or to indicate to a station a
set of preferred APs. RRM: Radio measurements enable STAs to understand the radio environment, it enables STAs to observe and gather data on radio link performance and on the radio environment. Current implementation adds beacon report, link measurement, neighbor report.

**Default value:**
- No (disabled)

**CONFIG_ESP_WIFI_SCAN_CACHE**

Keep scan results in cache

*Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_11KV_SUPPORT*

Keep scan results in cache, if not enabled, those will be flushed immediately.

**Default value:**
- No (disabled) if CONFIG_ESP_WIFI_11KV_SUPPORT

**CONFIG_ESP_WIFI_MBO_SUPPORT**

Enable Multi Band Operation Certification Support

*Found in: Component config > Wi-Fi*

Select this option to enable WiFi Multiband operation certification support.

**Default value:**
- No (disabled)

**CONFIG_ESP_WIFI_DPP_SUPPORT**

Enable DPP support

*Found in: Component config > Wi-Fi*

Select this option to enable WiFi Easy Connect Support.

**Default value:**
- No (disabled)

**CONFIG_ESP_WIFI_11R_SUPPORT**

Enable 802.11R (Fast Transition) Support

*Found in: Component config > Wi-Fi*

Select this option to enable WiFi Fast Transition Support.

**Default value:**
- No (disabled)

**CONFIG_ESP_WIFI_WPS_SOFTAP_REGISTRAR**

Add WPS Registrar support in SoftAP mode

*Found in: Component config > Wi-Fi*

Select this option to enable WPS registrar support in softAP mode.

**Default value:**
- No (disabled)
CONFIG_ESP_WIFI_ENABLE_WIFI_RX_STATS

Enable Wi-Fi reception statistics

Found in: Component config > Wi-Fi

Enable Wi-Fi reception statistics. Total support 2 access category. Each access category will use 190 bytes memory.

Default value:
- Yes (enabled) if SOC_WIFI_HE_SUPPORT

CONFIG_ESP_WIFI_ENABLE_WIFI_RX_MU_STATS

Enable Wi-Fi DL MU-MIMO and DL OFDMA reception statistics

Found in: Component config > Wi-Fi > CONFIG_ESP_WIFI_ENABLE_WIFI_RX_STATS

Enable Wi-Fi DL MU-MIMO and DL OFDMA reception statistics. Will use 10932 bytes memory.

Default value:
- Yes (enabled) if CONFIG_ESP_WIFI_ENABLE_WIFI_RX_STATS

WPS Configuration Options

Contains:
- CONFIG_ESP_WIFI_WPS_PASSPHRASE
- CONFIG_ESP_WIFI_WPS_STRICT

CONFIG_ESP_WIFI_WPS_STRICT

Strictly validate all WPS attributes

Found in: Component config > Wi-Fi > WPS Configuration Options

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.

Default value:
- No (disabled)

CONFIG_ESP_WIFI_WPS_PASSPHRASE

Get WPA2 passphrase in WPS config

Found in: Component config > Wi-Fi > WPS Configuration Options

Select this option to get passphrase during WPS configuration. This option fakes the virtual display capabilities to get the configuration in passphrase mode. Not recommended to be used since WPS credentials should not be shared to other devices, making it in readable format increases that risk, also passphrase requires pbkdf2 to convert in psk.

Default value:
- No (disabled)

CONFIG_ESP_WIFI_DEBUG_PRINT

Print debug messages from WPA Supplicant

Found in: Component config > Wi-Fi

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.

Default value:
CONFIG_ESP_WIFI_TESTING_OPTIONS

Add DPP testing code

*Found in: Component config > Wi-Fi*

Select this to enable unity test for DPP.

**Default value:**

- No (disabled)

CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT

Enable enterprise option

*Found in: Component config > Wi-Fi*

Select this to enable/disable enterprise connection support.

disabling this will reduce binary size. disabling this will disable the use of any esp_wifi_sta_wpa2_ent_*
(as APIs will be meaningless)

**Default value:**

- Yes (enabled)

Core dump  Contains:

- CONFIG_ESP_COREDUMP_CHECK_BOOT
- CONFIG_ESP_COREDUMP_DATA_FORMAT
- CONFIG_ESP_COREDUMP_CHECKSUM
- CONFIG_ESP_COREDUMP_TO_FLASH_OR_UART
- CONFIG_ESP_COREDUMP_UART_DELAY
- CONFIG_ESP_COREDUMP_FLASH_NO_OVERWRITE
- CONFIG_ESP_COREDUMP_LOGS
- CONFIG_ESP_COREDUMP_DECODE
- CONFIG_ESP_COREDUMP_MAX_TASKS_NUM
- CONFIG_ESP_COREDUMP_STACK_SIZE

CONFIG_ESP_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 compo-
nents/partition_table directory.

**Available options:**

- Flash (CONFIG_ESP_COREDUMP_ENABLE_TO_FLASH)
- UART (CONFIG_ESP_COREDUMP_ENABLE_TO_UART)
- None (CONFIG_ESP_COREDUMP_ENABLE_TO_NONE)
**CONFIG_ESP_COREDUMP_DATA_FORMAT**

Core dump data format

*Found in: Component config > Core dump*

Select the data format for core dump.

Available options:

- Binary format (CONFIG_ESP_COREDUMP_DATA_FORMAT_BIN)
- ELF format (CONFIG_ESP_COREDUMP_DATA_FORMAT_ELF)

**CONFIG_ESP_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 (CONFIG_ESP_COREDUMP_CHECKSUM_CRC32)
- Use SHA256 for integrity verification (CONFIG_ESP_COREDUMP_CHECKSUM_SHA256)

**CONFIG_ESP_COREDUMP_CHECK_BOOT**

Check core dump data integrity on boot

*Found in: Component config > Core dump*

When enabled, if any data are found on the flash core dump partition, they will be checked by calculating their checksum.

**Default value:**

- Yes (enabled) if CONFIG_ESP_COREDUMP_ENABLE_TO_FLASH

**CONFIG_ESP_COREDUMP_LOGS**

Enable coredump logs for debugging

*Found in: Component config > Core dump*

Enable/disable coredump logs. Logs strings from espcoredump component are placed in DRAM. Disabling these helps to save ~5KB of internal memory.

**CONFIG_ESP_COREDUMP_MAX_TASKS_NUM**

Maximum number of tasks

*Found in: Component config > Core dump*

Maximum number of tasks snapshots in core dump.

**CONFIG_ESP_COREDUMP_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.
Chapter 2. API Reference

Default value:
- 0 if `CONFIG_ESP_COREDUMP_ENABLE_TO_UART`

**CONFIG_ESP_COREDUMP_FLASH_NO_OVERWRITE**

Don’t overwrite existing core dump

*Found in: Component config > Core dump*

Don’t overwrite an existing core dump already present in flash. Enable this option to only keep the first of multiple core dumps.

If enabled, the core dump partition must be erased before the first core dump can be written.

Default value:
- No (disabled) if `CONFIG_ESP_COREDUMP_ENABLE_TO_FLASH`

**CONFIG_ESP_COREDUMP_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.

**CONFIG_ESP_COREDUMP_DECODE**

Handling of UART core dumps in IDF Monitor

*Found in: Component config > Core dump*

Available options:

- Decode and show summary (info_corefile) (CONFIG_ESP_COREDUMP_DECODE_INFO)
- Don’t decode (CONFIG_ESP_COREDUMP_DECODE_DISABLE)

**FAT Filesystem support**

Contains:

- `CONFIG_FATFS_API_ENCODING`
- `CONFIG_FATFS_VFS_FSTAT_BLKSIZE`
- `CONFIG_FATFS_IMMEDIATE_FSYNC`
- `CONFIG_FATFS_USE_FASTSEEK`
- `CONFIG_FATFS_LONG_FILENAMES`
- `CONFIG_FATFS_MAX_LFN`
- `CONFIG_FATFS_FS_LOCK`
- `CONFIG_FATFS_VOLUME_COUNT`
- `CONFIG_FATFS_CHOOSE_CODEPAGE`
- `CONFIG_FATFS_ALLOC_PREFER_EXTRAM`
- `CONFIG_FATFS_TIMEOUT_MS`
- `CONFIG_FATFS_PER_FILE_CACHE`

**CONFIG_FATFS_VOLUME_COUNT**
Number of volumes

*Found in: Component config > FAT Filesystem support*

Number of volumes (logical drives) to use.

**Range:**
- from 1 to 10

**Default value:**
- 2

**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 (CONFIG_FATFS_LFN_NONE)
- Long filename buffer in heap (CONFIG_FATFS_LFN_HEAP)
- Long filename buffer on stack (CONFIG_FATFS_LFN_STACK)

**CONFIG_FATFS_SECTOR_SIZE**

Sector size

*Found in: Component config > FAT Filesystem support*

Specify the size of the sector in bytes for FATFS partition generator.

Available options:

- 512 (CONFIG_FATFS_SECTOR_512)
- 4096 (CONFIG_FATFS_SECTOR_4096)

**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) (CONFIG_FATFS_CODEPAGE_DYNAMIC)
- US (CP437) (CONFIG_FATFS_CODEPAGE_437)
- Arabic (CP720) (CONFIG_FATFS_CODEPAGE_720)
- Greek (CP737) (CONFIG_FATFS_CODEPAGE_737)
- KBL (CP771) (CONFIG_FATFS_CODEPAGE_771)
- Baltic (CP775) (CONFIG_FATFS_CODEPAGE_775)
- Latin 1 (CP850) (CONFIG_FATFS_CODEPAGE_850)
- Latin 2 (CP852) (CONFIG_FATFS_CODEPAGE_852)
- Cyrillic (CP855) (CONFIG_FATFS_CODEPAGE_855)
Chapter 2. API Reference

- Turkish (CP857) (CONFIG_FATFS_CODEPAGE_857)
- Portuguese (CP860) (CONFIG_FATFS_CODEPAGE_860)
- Icelandic (CP861) (CONFIG_FATFS_CODEPAGE_861)
- Hebrew (CP862) (CONFIG_FATFS_CODEPAGE_862)
- Canadian French (CP863) (CONFIG_FATFS_CODEPAGE_863)
- Arabic (CP864) (CONFIG_FATFS_CODEPAGE_864)
- Nordic (CP865) (CONFIG_FATFS_CODEPAGE_865)
- Russian (CP866) (CONFIG_FATFS_CODEPAGE_866)
- Greek 2 (CP869) (CONFIG_FATFS_CODEPAGE_869)
- Japanese (DBCS) (CP932) (CONFIG_FATFS_CODEPAGE_932)
- Simplified Chinese (DBCS) (CP936) (CONFIG_FATFS_CODEPAGE_936)
- Korean (DBCS) (CP949) (CONFIG_FATFS_CODEPAGE_949)
- Traditional Chinese (DBCS) (CP950) (CONFIG_FATFS_CODEPAGE_950)

**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 (CONFIG_FATFS_API_ENCODING_ANSI_OEM)
- API uses UTF-8 encoding (CONFIG_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.

**Range:**

- from 0 to 65535

**Default value:**

- 0
**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 timeout after amount of time set by this option.

**Default value:**
- 10000

**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.

**Default value:**
- Yes (enabled)

**CONFIG_FATFS_ALLOC_PREFER_EXTRAM**

Perfer 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.

**Default value:**
- Yes (enabled) if CONFIG_SPIRAM_USE_CAPS_ALLOC || CONFIG_SPIRAM_USE_MALLOC

**CONFIG_FATFS_USE_FASTSEEK**

Enable fast seek algorithm when using lseek function through VFS FAT

*Found in: Component config > FAT Filesystem support*

The fast seek feature enables fast backward/long seek operations without FAT access by using an in-memory CLMT (cluster link map table). Please note, fast-seek is only allowed for read-mode files, if a file is opened in write-mode, the seek mechanism will automatically fallback to the default implementation.

**Default value:**
- No (disabled)
CONFIG_FATFS_FASTSEEK_BUFFER_SIZE

Fast seek CLMT buffer size

*Found in: Component config > FAT Filesystem support > CONFIG_FATFS_USE_FASTSEEK*

If fast seek algorithm is enabled, this defines the size of CLMT buffer used by this algorithm in 32-bit word units. This value should be chosen based on prior knowledge of maximum elements of each file entry would store.

**Default value:**
- 64 if CONFIG_FATFS_USE_FASTSEEK

CONFIG_FATFS_VFS_FSTAT_BLKSIZE

Default block size

*Found in: Component config > FAT Filesystem support*

If set to 0, the 'newlib' library's default size (BLKSIZ) is used (128 B). If set to a non-zero value, the value is used as the block size. Default file buffer size is set to this value and the buffer is allocated when first attempt of reading/writing to a file is made. Increasing this value improves fread() speed, however the heap usage is increased as well.

**NOTE:** The block size value is shared by all the filesystem functions accessing target media for given file descriptor! See 'Improving I/O performance' section of 'Maximizing Execution Speed' documentation page for more details.

**Default value:**
- 0

CONFIG_FATFS_IMMEDIATE_FSYNC

Enable automatic f_sync

*Found in: Component config > FAT Filesystem support*

Enables automatic calling of f_sync() to flush recent file changes after each call of vfs_fat_write(), vfs_fat_pwrite(), vfs_fat_link(), vfs_fat_truncate() and vfs_fat_ftruncate() functions. This feature improves file-consistency and size reporting accuracy for the FatFS, at a price on decreased performance due to frequent disk operations.

**Default value:**
- No (disabled)

FreeRTOS Contains:

- **Kernel**
- **Port**

Kernel Contains:

- **CONFIG_FREERTOS_CHECK_STACKOVERFLOW**
- **CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY**
- **CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS**
- **CONFIG_FREERTOS_MAX_TASK_NAME_LEN**
- **CONFIG_FREERTOS_IDLE_TASK_STACKSIZE**
- **CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS**
- **CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE**
- **CONFIG_FREERTOS_TASK_NOTIFICATION_ARRAY_ENTRIES**
- **CONFIG_FREERTOS_HZ**
- **CONFIG_FREERTOS_TIMER_QUEUE_LENGTH**
- **CONFIG_FREERTOS_TIMER_SERVICE_TASK_NAME**
Chapter 2. API Reference

- CONFIG_FREERTOS_TIMER_TASK_PRIORITY
- CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH
- CONFIG_FREERTOS_USE_IDLE_HOOK
- CONFIG_FREERTOS_USE_LIST_DATA_INTEGRITY_CHECK_BYTES
- CONFIG_FREERTOS_OPTIMIZED_SCHEDULER
- CONFIG_FREERTOS_USE_TICK_HOOK
- CONFIG_FREERTOS_USE_TICKLESS_IDLE
- CONFIG_FREERTOS_USE_TRACE_FACILITY
- CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID
- CONFIG_FREERTOS_UNICORE
- CONFIG_FREERTOS_SMP
- CONFIG_FREERTOS_USE_MINIMAL_IDLE_HOOK

CONFIG_FREERTOS_SMP

Run the Amazon SMP FreeRTOS kernel instead (FEATURE UNDER DEVELOPMENT)

Found in: Component config > FreeRTOS > Kernel

Amazon has released an SMP version of the FreeRTOS Kernel which can be found via the following link: https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/smp

IDF has added an experimental port of this SMP kernel located in components/freertos/FreeRTOS-Kernel-SMP. Enabling this option will cause IDF to use the Amazon SMP kernel. Note that THIS FEATURE IS UNDER ACTIVE DEVELOPMENT, users use this at their own risk.

Leaving this option disabled will mean the IDF FreeRTOS kernel is used instead, which is located in: components/freertos/FreeRTOS-Kernel. Both kernel versions are SMP capable, but differ in their implementation and features.

Default value:
- No (disabled)

CONFIG_FREERTOS_UNICORE

Run FreeRTOS only on first core

Found in: Component config > FreeRTOS > Kernel

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.

Default value:
- Yes (enabled)

CONFIG_FREERTOS_HZ

configTICK_RATE_HZ

Found in: Component config > FreeRTOS > Kernel

Sets the FreeRTOS tick interrupt frequency in Hz (see configTICK_RATE_HZ documentation for more details).

Range:
- from 1 to 1000

Default value:
- 100
**CONFIG_FREERTOS_OPTIMIZED_SCHEDULER**

```c
configUSE_PORT_OPTIMISED_TASK_SELECTION
```

*Found in: Component config > FreeRTOS > Kernel*

Enables port specific task selection method. This option can speed up the search of ready tasks when scheduling (see `configUSE_PORT_OPTIMISED_TASK_SELECTION` documentation for more details).

**CONFIG_FREERTOS_CHECK_STACKOVERFLOW**

```c
configCHECK_FOR_STACK_OVERFLOW
```

*Found in: Component config > FreeRTOS > Kernel*

Enables FreeRTOS to check for stack overflows (see `configCHECK_FOR_STACK_OVERFLOW` documentation for more details).

**Note:** If users do not provide their own `vApplicationStackOverflowHook()` function, a default function will be provided by ESP-IDF.

Available options:

- No checking (`CONFIG_FREERTOS_CHECK_STACKOVERFLOW_NONE`)
  Do not check for stack overflows (`configCHECK_FOR_STACK_OVERFLOW = 0`)
- Check by stack pointer value (Method 1) (`CONFIG_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 (Method 2) (`CONFIG_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 THREAD_LOCAL_STORAGE_POINTERS**

```c
configNUM_THREAD_LOCAL_STORAGE_POINTERS
```

*Found in: Component config > FreeRTOS > Kernel*

Set the number of thread local storage pointers in each task (see `configNUM_THREAD_LOCAL_STORAGE_POINTERS` documentation for more details).

**Note:** In ESP-IDF, 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.

**Range:**

- from 1 to 256

**Default value:**

- 1

**CONFIG_FREERTOS_IDLE_TASK_STACKSIZE**

```c
configMINIMAL_STACK_SIZE (Idle task stack size)
```

*Found in: Component config > FreeRTOS > Kernel*

Sets the idle task stack size in bytes (see `configMINIMAL_STACK_SIZE` documentation for more details).

**Note:**
ESP-IDF specifies stack sizes in bytes instead of words.

The default size is enough for most use cases.

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.

Conversely, the stack size can be reduced to the minimum if none of the idle features are used.

**Range:**
- from 768 to 32768

**Default value:**
- 1536

---

**CONFIG_FREERTOS_USE_IDLE_HOOK**

`configUSE_IDLE_HOOK`

*Found in: Component config > FreeRTOS > Kernel*

Enables the idle task application hook (see `configUSE_IDLE_HOOK` documentation for more details).

**Note:**
- The application must provide the hook function `void vApplicationIdleHook( void );`
- `vApplicationIdleHook()` is called from FreeRTOS idle task(s)
- The FreeRTOS idle hook is NOT the same as the ESP-IDF Idle Hook, but both can be enabled simultaneously.

**Default value:**
- No (disabled)

---

**CONFIG_FREERTOS_USE_MINIMAL_IDLE_HOOK**

Use `FreeRTOS` minimal idle hook

*Found in: Component config > FreeRTOS > Kernel*

Enables the minimal idle task application hook (see `configUSE_IDLE_HOOK` documentation for more details).

**Note:**
- The application must provide the hook function `void vApplicationMinimalIdleHook( void );`
- `vApplicationMinimalIdleHook()` is called from FreeRTOS minimal idle task(s)

**Default value:**
- No (disabled) if `CONFIG_FREERTOS_SMP`

---

**CONFIG_FREERTOS_USE_TICK_HOOK**

`configUSE_TICK_HOOK`

*Found in: Component config > FreeRTOS > Kernel*

Enables the tick hook (see `configUSE_TICK_HOOK` documentation for more details).

**Note:**
- The application must provide the hook function `void vApplicationTickHook( void );`
- `vApplicationTickHook()` is called from FreeRTOS's tick handling function `xTaskIncrementTick()`
- The FreeRTOS tick hook is NOT the same as the ESP-IDF Tick Interrupt Hook, but both can be enabled simultaneously.
CONFIG_FREERTOS_MAX_TASK_NAME_LEN

configMAX_TASK_NAME_LEN

*Found in: Component config > FreeRTOS > Kernel*

Sets the maximum number of characters for task names (see configMAX_TASK_NAME_LEN documentation for more details).

*Note: For most uses, the default of 16 characters is sufficient.*

*Range:*
  * from 1 to 256

*Default value:*
  * 16

CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY

configENABLE_BACKWARD_COMPATIBILITY

*Found in: Component config > FreeRTOS > Kernel*

Enable backward compatibility with APIs prior to FreeRTOS v8.0.0. (see configENABLE_BACKWARD_COMPATIBILITY documentation for more details).

*Default value:*
  * No (disabled)

CONFIG_FREERTOS_TIMER_SERVICE_TASK_NAME

configTIMER_SERVICE_TASK_NAME

*Found in: Component config > FreeRTOS > Kernel*

Sets the timer task’s name (see configTIMER_SERVICE_TASK_NAME documentation for more details).

*Default value:*
  * “Tmr Svc”

CONFIG_FREERTOS_TIMER_TASK_PRIORITY

configTIMER_TASK_PRIORITY

*Found in: Component config > FreeRTOS > Kernel*

Sets the timer task’s priority (see configTIMER_TASK_PRIORITY documentation for more details).

*Range:*
  * from 1 to 25

*Default value:*
  * 1

CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH

configTIMER_TASK_STACK_DEPTH

*Found in: Component config > FreeRTOS > Kernel*

Set the timer task’s stack size (see configTIMER_TASK_STACK_DEPTH documentation for more details).
CONFIG_FREERTOS_TIMER_QUEUE_LENGTH

`configTIMER_QUEUE_LENGTH`

*Found in: Component config > FreeRTOS > Kernel*

Set the timer task’s command queue length (see `configTIMER_QUEUE_LENGTH` documentation for more details).

**Range:**
- from 5 to 20

**Default value:**
- 10

CONFIG_FREERTOS_QUEUE_REGISTRY_SIZE

`configQUEUE_REGISTRY_SIZE`

*Found in: Component config > FreeRTOS > Kernel*

Set the size of the queue registry (see `configQUEUE_REGISTRY_SIZE` documentation for more details).

Note: A value of 0 will disable queue registry functionality

**Range:**
- from 0 to 20

**Default value:**
- 0

CONFIG_FREERTOS_TASK_NOTIFICATION_ARRAY_ENTRIES

`configTASK_NOTIFICATION_ARRAY_ENTRIES`

*Found in: Component config > FreeRTOS > Kernel*

Set the size of the task notification array of each task. When increasing this value, keep in mind that this means additional memory for each and every task on the system. However, task notifications in general are more lightweight compared to alternatives such as semaphores.

**Range:**
- from 1 to 32

**Default value:**
- 1

CONFIG_FREERTOS_USE_TRACE_FACILITY

`configUSE_TRACE_FACILITY`

*Found in: Component config > FreeRTOS > Kernel*

Enables additional structure members and functions to assist with execution visualization and tracing (see `configUSE_TRACE_FACILITY` documentation for more details).

**Default value:**
- No (disabled)
**CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS**

`configUSE_STATS_FORMATTING_FUNCTIONS`

*Found in: Component config > FreeRTOS > Kernel > CONFIG_FREERTOS_USE_TRACE_FACILITY*

Set `configUSE_TRACE_FACILITY` and `configUSE_STATS_FORMATTING_FUNCTIONS` to 1 to include the `vTaskList()` and `vTaskGetRunTimeStats()` functions in the build (see `configUSE_STATS_FORMATTING_FUNCTIONS` documentation for more details).

**Default value:**
- No (disabled) if `CONFIG_FREERTOS_USE_TRACE_FACILITY`

**CONFIG_FREERTOS_USE_LIST_DATA_INTEGRITY_CHECK_BYTES**

`configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES`

*Found in: Component config > FreeRTOS > Kernel*

Enable list integrity checker (see `configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES` documentation for more details).

**Default value:**
- No (disabled)

**CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID**

Enable display of xCoreID in vTaskList

*Found in: Component config > FreeRTOS > Kernel*

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**

`configGENERATE_RUN_TIME_STATS`

*Found in: Component config > FreeRTOS > Kernel*

Enables collection of run time statistics for each task (see `configGENERATE_RUN_TIME_STATS` documentation for more details).

**Note:** The clock used for run time statistics can be configured in `FREERTOS_RUN_TIME_STATS_CLK`.

**Default value:**
- No (disabled)

**CONFIG_FREERTOS_RUN_TIME_COUNTER_TYPE**

`configRUN_TIME_COUNTER_TYPE`

*Found in: Component config > FreeRTOS > Kernel > CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS*

Sets the data type used for the FreeRTOS run time stats. A larger data type can be used to reduce the frequency of the counter overflowing.

**Available options:**
- `uint32_t` (CONFIG_FREERTOS_RUN_TIME_COUNTER_TYPE_U32) `configRUN_TIME_COUNTER_TYPE` is set to `uint32_t`
**CONFIG_FREERTOS_USE_TICKLESS_IDLE**

```c
configUSE_TICKLESS_IDLE
```

*Found in: Component config > FreeRTOS > Kernel*

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. To skip unnecessary wake-up initialize a timer with the “skip_unhandled_events” option as true.

If disabled, automatic light sleep support will be disabled.

**Default value:**
- No (disabled) if `CONFIG_PM_ENABLE`

**CONFIG_FREERTOS_IDLE_TIME_BEFORE_SLEEP**

```c
configEXPECTED_IDLE_TIME_BEFORE_SLEEP
```

*Found in: Component config > FreeRTOS > Kernel > CONFIG_FREERTOS_USE_TICKLESS_IDLE*

FreeRTOS will enter light sleep mode if no tasks need to run for this number of ticks. You can enable `PM_PROFILING` feature in `esp_pm` components and dump the sleep status with `esp_pm_dump_locks`, if the proportion of rejected sleeps is too high, please increase this value to improve scheduling efficiency.

**Range:**
- from 2 to 4294967295 if `CONFIG_FREERTOS_USE_TICKLESS_IDLE`

**Default value:**
- 3 if `CONFIG_FREERTOS_USE_TICKLESS_IDLE`

**Port**

Contains:

- `CONFIG_FREERTOS_CHECK_MUTEX_GIVEN_BY_OWNER`
- `CONFIG_FREERTOS_RUN_TIME_STATS_CLK`
- `CONFIG_FREERTOS_INTERRUPT_BACKTRACE`
- `CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK`
- `CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP`
- `CONFIG_FREERTOS_TASK_PRE_DELETION_HOOK`
- `CONFIG_FREERTOS_TLSP_DELETION_CALLBACKS`
- `CONFIG_FREERTOS_ISR_STACKSIZE`
- `CONFIG_FREERTOSPLACE_FUNCTIONS_INTO_FLASH`
- `CONFIG_FREERTOS_CHECK_PORT_CRITICAL_COMPLIANCE`
- `CONFIG_FREERTOS_CORETIMER`
- `CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER`

**CONFIG_FREERTOS_TASK_FUNCTION_WRAPPER**

Wrap task functions

*Found in: Component config > FreeRTOS > Port*

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.

**Default value:**
- Yes (enabled)

**CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK**

Enable stack overflow debug watchpoint

*Found in: Component config > FreeRTOS > Port*

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 last debug memory watchpoint 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 have one hardware watchpoint less because the last one is overwritten as soon as a task switch happens.

Another consequence is that due to alignment requirements of the watchpoint, the usable stack size decreases by up to 60 bytes. This is because the watchpoint region has to be aligned to its size and the size for the stack watchpoint in IDF is 32 bytes.

This check only triggers if the stack overflow writes within 32 bytes near 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.

**Default value:**
- No (disabled)

**CONFIG_FREERTOS_TLSP_DELETION_CALLBACKS**

Enable thread local storage pointers deletion callbacks

*Found in: Component config > FreeRTOS > Port*

ESP-IDF provides users with the ability to free TLSP memory by registering TLSP deletion callbacks. These callbacks are automatically called by FreeRTOS when a task is deleted. When this option is turned on, the memory reserved for TLSPs in the TCB is doubled to make space for storing the deletion callbacks. If the user does not wish to use TLSP deletion callbacks then this option could be turned off to save space in the TCB memory.

**Default value:**
- Yes (enabled)

**CONFIG_FREERTOS_TASK_PRE_DELETION_HOOK**

Enable task pre-deletion hook

*Found in: Component config > FreeRTOS > Port*

Enable this option to make FreeRTOS call a user provided hook function right before it deletes a task (i.e., frees/releases a dynamically/statically allocated task’s memory). This is useful if users want to know when a task is actually deleted (in case the task’s deletion is delegated to the IDLE task).

If this config option is enabled, users must define a `void vTaskPreDeletionHook( void * pxTCB )` hook function in their application.
CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP
Enable static task clean up hook (DEPRECATED)

*Found in: Component config > FreeRTOS > Port*

THIS OPTION IS DEPRECATED. Use FREERTOS_TASK_PRE_DELETION_HOOK instead.

Enable this option to make FreeRTOS call the static task clean up hook when a task is deleted.

Note: Users will need to provide a `void vPortCleanUpTCB ( void *pxTCB )` callback

**Default value:**
- No (disabled)

CONFIG_FREERTOS_CHECK_MUTEX_GIVEN_BY_OWNER
Check that mutex semaphore is given by owner task

*Found in: Component config > FreeRTOS > Port*

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_ISR_STACKSIZE
ISR stack size

*Found in: Component config > FreeRTOS > Port*

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.

**Range:**
- from 2096 to 32768 if `CONFIG_ESP_COREDUMP_DATA_FORMAT_ELF`
- from 1536 to 32768

**Default value:**
- 2096 if `CONFIG_ESP_COREDUMP_DATA_FORMAT_ELF`
- 1536

CONFIG_FREERTOS_INTERRUPT_BACKTRACE
Enable backtrace from interrupt to task context

*Found in: Component config > FreeRTOS > Port*

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.

**Default value:**
- Yes (enabled)

CONFIG_FREERTOS_CORETIMER
Tick timer source (Xtensa Only)

*Found in: Component config > FreeRTOS > Port*

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.

Available options:
- Timer 0 (int 6, level 1) (CONFIG_FREERTOS_CORETIMER_0)
  Select this to use timer 0
- Timer 1 (int 15, level 3) (CONFIG_FREERTOS_CORETIMER_1)
  Select this to use timer 1
- SYSTIMER 0 (level 1) (CONFIG_FREERTOS_CORETIMER_SYSTIMER_LVL1)
  Select this to use systimer with the 1 interrupt priority.
- SYSTIMER 0 (level 3) (CONFIG_FREERTOS_CORETIMER_SYSTIMER_LVL3)
  Select this to use systimer with the 3 interrupt priority.

**CONFIG_FREERTOS_RUN_TIME_STATS_CLK**

Choose the clock source for run time stats

*Found in: Component config > FreeRTOS > Port*

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 (CONFIG_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 (CONFIG_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 ESP_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_PLACE_FUNCTIONS_INTO_FLASH**

Place FreeRTOS functions into Flash

*Found in: Component config > FreeRTOS > Port*

When enabled these selected Non-ISR FreeRTOS functions will be placed into Flash memory instead of IRAM. This saves up to 8KB of IRAM depending on which functions are used.

**Default value:**

- No (disabled)

**CONFIG_FREERTOS_CHECK_PORT_CRITICAL_COMPLIANCE**

Tests compliance with Vanilla FreeRTOS port*._CRITICAL calls

*Found in: Component config > FreeRTOS > Port*

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

**Default value:**

- No (disabled)
Hardware Abstraction Layer (HAL) and Low Level (LL) Contains:

- `CONFIG_HAL_DEFAULT_ASSERTION_LEVEL`
- `CONFIG_HAL_LOG_LEVEL`
- `CONFIG_HAL_SYSTIMER_USE_ROM_IMPL`
- `CONFIG_HAL_WDT_USE_ROM_IMPL`

**CONFIG_HAL_DEFAULT_ASSERTION_LEVEL**

Default HAL assertion level

*Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)*

Set the assert behavior / level for HAL component. HAL component assert level can be set separately, but the level can’t exceed the system assertion level. E.g. If the system assertion is disabled, then the HAL assertion can’t be enabled either. If the system assertion is enable, then the HAL assertion can still be disabled by this Kconfig option.

Available options:

- Same as system assertion level (`CONFIG_HAL_ASSERTION_EQUALS_SYSTEM`)
- Disabled (`CONFIG_HAL_ASSERTION_DISABLE`)
- Silent (`CONFIG_HAL_ASSERTION_SILENT`)
- Enabled (`CONFIG_HAL_ASSERTION_ENABLE`)

**CONFIG_HAL_LOG_LEVEL**

HAL layer log verbosity

*Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)*

Specify how much output to see in HAL logs.

Available options:

- No output (`CONFIG_HAL_LOG_LEVEL_NONE`)
- Error (`CONFIG_HAL_LOG_LEVEL_ERROR`)
- Warning (`CONFIG_HAL_LOG_LEVEL_WARN`)
- Info (`CONFIG_HAL_LOG_LEVEL_INFO`)
- Debug (`CONFIG_HAL_LOG_LEVEL_DEBUG`)
- Verbose (`CONFIG_HAL_LOG_LEVEL_VERBOSE`)

**CONFIG_HAL_SYSTIMER_USE_ROM_IMPL**

Use ROM implementation of SysTimer HAL driver

*Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)*

Enable this flag to use HAL functions from ROM instead of ESP-IDF.

If keeping this as “n” in your project, you will have less free IRAM. If making this as “y” in your project, you will increase free IRAM, but you will lose the possibility to debug this module, and some new features will be added and bugs will be fixed in the IDF source but cannot be synced to ROM.

Default value:

- Yes (enabled) if `ESP_ROM_HAS_HAL_SYSTIMER`
CONFIG_HAL_WDT_USE_ROMImpl

Use ROM implementation of WDT HAL driver

*Found in: Component config > Hardware Abstraction Layer (HAL) and Low Level (LL)*

Enable this flag to use HAL functions from ROM instead of ESP-IDF.

If keeping this as “n” in your project, you will have less free IRAM. If making this as “y” in your project, you will increase free IRAM, but you will lose the possibility to debug this module, and some new features will be added and bugs will be fixed in the IDF source but cannot be synced to ROM.

**Default value:**
- Yes (enabled) if ESP_ROM_HAS_HAL_WDT

Heap memory debugging  Contains:

- CONFIG_HEAP_ABORT_WHEN_ALLOCATION_FAILS
- CONFIG_HEAP_TASK_TRACKING
- CONFIG_HEAP_PLACE_FUNCTION_INTO_FLASH
- CONFIG_HEAP_CORRUPTION_DETECTION
- CONFIG_HEAP_TRACING_DEST
- CONFIG_HEAP_TRACING_STACK_DEPTH
- CONFIG_HEAP_USE_HOOKS
- CONFIG_HEAP_TRACE_HASH_MAP
- CONFIG_HEAP_TLSF_USE_ROM_IMPL

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) (CONFIG HEAP POISONING DISABLED)
- Light impact (CONFIG_HEAP_POISONING_LIGHT)
- Comprehensive (CONFIG_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 side 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 (CONFIG_HEAP_TRACING_OFF)
- Standalone (CONFIG_HEAP_TRACING_STANDALONE)
- Host-based (CONFIG_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_USE_HOOKS**

Use allocation and free hooks

*Found in: Component config > Heap memory debugging*

Enable the user to implement function hooks triggered for each successful allocation and free.

**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.

**CONFIG_HEAP_TRACE_HASH_MAP**

Use hash map mechanism to access heap trace records

*Found in: Component config > Heap memory debugging*

Enable this flag to use a hash map to increase performance in handling heap trace records.

Heap trace standalone supports storing records as a list, or a list + hash map.

Using only a list takes less memory, but calls to `free` will get slower as the list grows. This is particularly affected when using `HEAP_TRACE_ALL` mode.

By using a list + hash map, calls to `free` remain fast, at the cost of additional memory to store the hash map.

**Default value:**

- No (disabled) if `CONFIG_HEAP_TRACING_STANDALONE`

**CONFIG_HEAP_TRACE_HASH_MAP_IN_EXT_RAM**

Place hash map in external RAM

*Found in: Component config > Heap memory debugging > CONFIG_HEAP_TRACE_HASH_MAP*

When enabled this configuration forces the hash map to be placed in external RAM.

**Default value:**

- No (disabled) if `CONFIG_HEAP_TRACE_HASH_MAP`
CONFIG_HEAP_TRACE_HASH_MAP_SIZE

The number of entries in the hash map

*Found in: Component config > Heap memory debugging > CONFIG_HEAP_TRACE_HASH_MAP*

Defines the number of entries in the heap trace hashmap. Each entry takes 8 bytes. The bigger this number is, the better the performance. Recommended range: 200 - 2000.

**Default value:**
- 512 if `CONFIG_HEAP_TRACE_HASH_MAP`

CONFIG_HEAP_ABORT_WHEN_ALLOCATION_FAILS

Abort if memory allocation fails

*Found in: Component config > Heap memory debugging*

When enabled, if a memory allocation operation fails it will cause a system abort.

**Default value:**
- No (disabled)

CONFIG_HEAP_TLSF_USE_ROM_IMPL

Use ROM implementation of heap tlsf library

*Found in: Component config > Heap memory debugging*

Enable this flag to use heap functions from ROM instead of ESP-IDF.

If keeping this as “n” in your project, you will have less free IRAM. If making this as “y” in your project, you will increase free IRAM, but you will lose the possibility to debug this module, and some new features will be added and bugs will be fixed in the IDF source but cannot be synced to ROM.

**Default value:**
- Yes (enabled) if `ESP_ROM_HAS_HEAP_TLSF`

CONFIG_HEAP_PLACE_FUNCTION_INTO_FLASH

Force the entire heap component to be placed in flash memory

*Found in: Component config > Heap memory debugging*

Enable this flag to save up RAM space by placing the heap component in the flash memory

Note that it is only safe to enable this configuration if no functions from esp_heap_caps.h or esp_heap_trace.h are called from ISR.

IEEE 802.15.4

Contains:
- `CONFIG_IEEE802154_CCA_THRESHOLD`
- `CONFIG_IEEE802154_CCA_MODE`
- `CONFIG_IEEE802154_DEBUG`
- `CONFIG_IEEE802154_SLEEP_ENABLE`
- `CONFIG_IEEE802154_MULTI_PAN_ENABLE`
- `CONFIG_IEEE802154_TIMING_OPTIMIZATION`
- `CONFIG_IEEE802154_PENDING_TABLE_SIZE`
- `CONFIG_IEEE802154_RX_BUFFER_SIZE`
**CONFIG_IEEE802154_RX_BUFFER_SIZE**

The number of 802.15.4 receive buffers

*Found in: Component config > IEEE 802.15.4*

The number of 802.15.4 receive buffers

**CONFIG_IEEE802154_CCA_MODE**

Clear Channel Assessment (CCA) mode

*Found in: Component config > IEEE 802.15.4*

c configure the CCA mode

Available options:

- Carrier sense only (CONFIG_IEEE802154_CCA_CARRIER)
  configure the CCA mode to Energy above threshold
- Energy above threshold (CONFIG_IEEE802154_CCA_ED)
  configure the CCA mode to Energy above threshold
- Carrier sense OR energy above threshold (CONFIG_IEEE802154_CCA_CARRIER_OR_ED)
  configure the CCA mode to Carrier sense OR energy above threshold
- Carrier sense AND energy above threshold (CONFIG_IEEE802154_CCA_CARRIER_AND_ED)
  configure the CCA mode to Carrier sense AND energy above threshold

**CONFIG_IEEE802154_CCA_THRESHOLD**

CCA detection threshold

*Found in: Component config > IEEE 802.15.4*

set the CCA threshold, in dB

**Range:**

- from -120 to 0

**Default value:**

- “-60”

**CONFIG_IEEE802154_PENDING_TABLE_SIZE**

Pending table size

*Found in: Component config > IEEE 802.15.4*

set the pending table size

**Range:**

- from 1 to 100

**Default value:**

- 20

**CONFIG_IEEE802154_MULTI_PAN_ENABLE**

Enable multi-pan feature for frame filter

*Found in: Component config > IEEE 802.15.4*

Enable IEEE802154 multi-pan

**Default value:**
Chapter 2. API Reference

• No (disabled)

**CONFIG_IEEE802154_TIMING_OPTIMIZATION**

Enable throughput optimization

*Found in: Component config > IEEE 802.15.4*

Enabling this option increases throughput by ~5% at the expense of ~2.1k IRAM code size increase.

**Default value:**

• No (disabled)

**CONFIG_IEEE802154_SLEEP_ENABLE**

Enable IEEE802154 light sleep

*Found in: Component config > IEEE 802.15.4*

Enabling this option allows the IEEE802.15.4 module to be powered down during automatic light sleep, which reduces current consumption.

**Default value:**

• No (disabled) if CONFIG_PM_ENABLE & & CONFIG_PM_POWER_DOWN_PERIPHERAL_IN_LIGHT_SLEEP

**CONFIG_IEEE802154_DEBUG**

Enable IEEE802154 Debug

*Found in: Component config > IEEE 802.15.4*

Enabling this option allows different kinds of IEEE802154 debug output. All IEEE802154 debug features increase the size of the final binary.

**Default value:**

• No (disabled)

Contains:

• CONFIG_IEEE802154_RECORD_ABORT
• CONFIG_IEEE802154_RECORD_CMD
• CONFIG_IEEE802154_RECORD_EVENT
• CONFIG_IEEE802154_RECORD_STATE
• CONFIG_IEEE802154_ASSERT

**CONFIG_IEEE802154_ASSERT**

Enrich the assert information with IEEE802154 state and event

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG*

Enabling this option to add some probe codes in the driver, and these informations will be printed when assert.

**Default value:**

• No (disabled) if CONFIG_IEEE802154_DEBUG

**CONFIG_IEEE802154_RECORD_EVENT**

Enable record event information for debugging

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG*

Enabling this option to record event, when assert, the recorded event will be printed.

**Default value:**
Chapter 2. API Reference

- No (disabled) if `CONFIG_IEEE802154_DEBUG`

**CONFIG_IEEE802154_RECORD_EVENT_SIZE**

Record event table size

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG > CONFIG_IEEE802154_RECORD_EVENT*

set the record event table size

**Range:**
- from 1 to 50 if `CONFIG_IEEE802154_RECORD_EVENT`

**Default value:**
- 30 if `CONFIG_IEEE802154_RECORD_EVENT`

**CONFIG_IEEE802154_RECORD_STATE**

Enable record state information for debugging

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG*

Enabling this option to record state, when assert, the recorded state will be printed.

**Default value:**
- No (disabled) if `CONFIG_IEEE802154_DEBUG`

**CONFIG_IEEE802154_RECORD_STATE_SIZE**

Record state table size

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG > CONFIG_IEEE802154_RECORD_STATE*

set the record state table size

**Range:**
- from 1 to 50 if `CONFIG_IEEE802154_RECORD_STATE`

**Default value:**
- 10 if `CONFIG_IEEE802154_RECORD_STATE`

**CONFIG_IEEE802154_RECORD_CMD**

Enable record command information for debugging

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG*

Enabling this option to record the command, when assert, the recorded command will be printed.

**Default value:**
- No (disabled) if `CONFIG_IEEE802154_DEBUG`

**CONFIG_IEEE802154_RECORD_CMD_SIZE**

Record command table size

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG > CONFIG_IEEE802154_RECORD_CMD*

set the record command table size

**Range:**
- from 1 to 50 if `CONFIG_IEEE802154_RECORD_CMD`

**Default value:**
- 10 if `CONFIG_IEEE802154_RECORD_CMD`
CONFIG_IEEE802154_RECORD_ABORT
Enable record abort information for debugging

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG*

Enabling this option to record the abort, when assert, the recorded abort will be printed.

**Default value:**
- No (disabled) if CONFIG_IEEE802154_DEBUG

CONFIG_IEEE802154_RECORD_ABORT_SIZE
Record abort table size

*Found in: Component config > IEEE 802.15.4 > CONFIG_IEEE802154_DEBUG > CONFIG_IEEE802154_RECORD_ABORT*

set the record abort table size

**Range:**
- from 1 to 50 if CONFIG_IEEE802154_RECORD_ABORT

**Default value:**
- 10 if CONFIG_IEEE802154_RECORD_ABORT

Log output  Contains:

- CONFIG_LOG_DEFAULT_LEVEL
- CONFIG_LOG_MASTER_LEVEL
- CONFIG_LOG_TIMESTAMP_SOURCE
- CONFIG_LOG_MAXIMUM_LEVEL
- CONFIG_LOG_COLORS

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.

By default, this setting limits which log statements are compiled into the program. For example, selecting “Warning” would mean that changing log level to “Debug” at runtime will not be possible. To allow increasing log level above the default at runtime, see the next option.

**Available options:**

- No output (CONFIG_LOG_DEFAULT_LEVEL_NONE)
- Error (CONFIG_LOG_DEFAULT_LEVEL_ERROR)
- Warning (CONFIG_LOG_DEFAULT_LEVEL_WARN)
- Info (CONFIG_LOG_DEFAULT_LEVEL_INFO)
- Debug (CONFIG_LOG_DEFAULT_LEVEL_DEBUG)
- Verbose (CONFIG_LOG_DEFAULT_LEVEL_VERBOSE)

CONFIG_LOG_MAXIMUM_LEVEL
Maximum log verbosity

*Found in: Component config > Log output*
This config option sets the highest log verbosity that it’s possible to select at runtime by calling `esp_log_level_set()`. This level may be higher than the default verbosity level which is set when the app starts up.

This can be used to enable debugging output only at a critical point, for a particular tag, or to minimize startup time but then enable more logs once the firmware has loaded.

Note that increasing the maximum available log level will increase the firmware binary size.

This option only applies to logging from the app, the bootloader log level is fixed at compile time to the separate “Bootloader log verbosity” setting.

Available options:

- Same as default (CONFIG_LOG_MAXIMUM_EQUALS_DEFAULT)
- Error (CONFIG_LOG_MAXIMUM_LEVEL_ERROR)
- Warning (CONFIG_LOG_MAXIMUM_LEVEL_WARN)
- Info (CONFIG_LOG_MAXIMUM_LEVEL_INFO)
- Debug (CONFIG_LOG_MAXIMUM_LEVEL_DEBUG)
- Verbose (CONFIG_LOG_MAXIMUM_LEVEL_VERBOSE)

**CONFIG_LOG_MASTER_LEVEL**

Enable global master log level

*Found in: Component config > Log output*

Enables an additional global “master” log level check that occurs before a log tag cache lookup. This is useful if you want to compile in a lot of logs that are selectable at runtime, but avoid the performance hit during periods where you don’t want log output. Examples include remote log forwarding, or disabling logs during a time-critical or CPU-intensive section and re-enabling them later. Results in larger program size depending on number of logs compiled in.

If enabled, defaults to LOG_DEFAULT_LEVEL and can be set using `esp_log_set_level_master()`. This check takes precedence over ESP_LOG_LEVEL_LOCAL.

**Default value:**
- No (disabled)

**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.

**Default value:**
- Yes (enabled)

**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 chip’s RTC and high resolution timers to maintain an accurate time. The system time is initialized to 0 on startup, it can be set with an NTP 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 (CONFIG_LOG_TIMESTAMP_SOURCE_RTOS)
• System Time (CONFIG_LOG_TIMESTAMP_SOURCE_SYSTEM)

**LWIP** Contains:

- CONFIG_LWIP_CHECK_THREAD_SAFETY
- Checksums
- CONFIG_LWIP_DHCP_COARSE_TIMER_SECS
- DHCP server
- CONFIG_LWIP_DHCP_OPTIONS_LEN
- CONFIG_LWIP_DHCP_DISABLE_CLIENT_ID
- CONFIG_LWIP_DHCP_DISABLE_VENDOR_CLASS_ID
- CONFIG_LWIP_DHCP_DOES_ARP_CHECK
- CONFIG_LWIP_DHCP_RESTORE_LAST_IP
- CONFIG_LWIP_PPP_CHAP_SUPPORT
- CONFIG_LWIP_L2_TO_L3_COPY
- CONFIG_LWIP_IPV6_DHCP6
- CONFIG_LWIP_IP4_FRAG
- CONFIG_LWIP_IP6_FRAG
- CONFIG_LWIP_IP_FORWARD
- CONFIG_LWIP_NETBUF_RECVINFO
- CONFIG_LWIP_IPV4
- CONFIG_LWIP_AUTOIP
- CONFIG_LWIP_IPV6
- CONFIG_LWIP_ENABLE_LCP_ECHO
- CONFIG_LWIP_ESP_LWIP_ASSERT
- CONFIG_LWIP_DEBUG
- CONFIG_LWIP_IRAM_OPTIMIZATION
- CONFIG_LWIP_EXTRA_IRAM_OPTIMIZATION
- CONFIG_LWIP_ENABLE
- CONFIG_LWIP_STATS
- CONFIG_LWIP_TIMERS_ONDEMAND
- CONFIG_LWIP_DNS_SUPPORT_MDNS QUERIES
- CONFIG_LWIP_PPP_MPPE_SUPPORT
- CONFIG_LWIP_PPP_MSCHAP_SUPPORT
- CONFIG_LWIP_PPP_NOTIFY PHASE_SUPPORT
- CONFIG_LWIP_PPP_PAP_SUPPORT
- CONFIG_LWIP_PPP_DEBUG_ON
- CONFIG_LWIP_PPP_SUPPORT
- CONFIG_LWIP_IP4_REASSEMBLY
- CONFIG_LWIP_IP6_REASSEMBLY
- CONFIG_LWIP_SLIP_SUPPORT
- CONFIG_LWIP_SO_LINGER
- CONFIG_LWIP_SO_RCVBUF
- CONFIG_LWIP_SO_REUSE
- CONFIG_LWIP_NETIF_STATUS_CALLBACK
- CONFIG_LWIP_TCPIP_CORE_LOCKING
- CONFIG_LWIP_NETIF_API
Chapter 2. API Reference

- **Hooks**
  - **ICMP**
  - **CONFIG_LWIP_LOCAL_HOSTNAME**
  - **CONFIG_LWIP_ND6**
  - **LWIP RAW API**
  - **CONFIG_LWIP_TCPPIP_TASK_PRIO**
  - **CONFIG_LWIP_IPV6_ND6_NUM_NEIGHBORS**
  - **CONFIG_LWIP_IPV6_MEMP_NUM_ND6_QUEUE**
  - **CONFIG_LWIP_MAX_SOCKETS**
  - **CONFIG_LWIP_BRIDGEIF_MAX_PORTS**
  - **CONFIG_LWIP_NUM_NETIF_CLIENT_DATA**
  - **CONFIG_LWIP_ESP Gratuitous ARP**
  - **CONFIG_LWIP_ESP_MLDV6_REPORT**
  - **SNTP**
  - **CONFIG_LWIP_USE_ONLY_LWIP_SELECT**
  - **CONFIG_LWIP_NETIF LOOPBACK**
  - **TCP**
  - **CONFIG_LWIP_TCPPIP TASK AFFINITY**
  - **CONFIG_LWIP_TCPPIP TASK STACK SIZE**
  - **CONFIG_LWIP_TCPPIP_RECVBOX_SIZE**
  - **CONFIG_LWIP_IPV6_REASS_MAX_PBUFS**
  - **UDP**
  - **CONFIG_LWIP_IPV6_RDNSS_MAX_DNS_SERVERS**

**CONFIG_LWIP_ENABLE**

Enable LwIP stack

*Found in: Component config > LWIP*

Builds normally if selected. Excludes LwIP from build if unselected, even if it is a dependency of a component or application. Some applications can switch their IP stacks, e.g., when switching between chip and Linux targets (LwIP stack vs. Linux IP stack). Since the LwIP dependency cannot easily be excluded based on a Kconfig option, it has to be a dependency in all cases. This switch allows the LwIP stack to be built selectively, even if it is a dependency.

**Default value:**

- Yes (enabled)

**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()

**Default value:**

- “espressif”

**CONFIG_LWIP_NETIF_API**

Enable usage of standard POSIX APIs in LWIP

*Found in: Component config > LWIP*

If this feature is enabled, standard POSIX APIs: if_indextoname(), if_nametoindex() could be used to convert network interface index to name instead of IDF specific esp-netif APIs (such as esp_netif_get_netif_impl_name())

**Default value:**
Chapter 2. API Reference

- No (disabled)

**CONFIG_LWIP_TCPIP_TASK_PRIO**

LWIP TCP/IP Task Priority

*Found in: Component config > LWIP*

LWIP tcpip task priority. In case of high throughput, this parameter could be changed up to (config-MAX_PRIORITIES-1).

*Range:*
  - from 1 to 24

*Default value:*
  - 18

**CONFIG_LWIP_TCPIP_CORE_LOCKING**

Enable tcpip core locking

*Found in: Component config > LWIP*

If Enable tcpip core locking, creates a global mutex that is held during TCPIP thread operations. Can be locked by client code to perform lwIP operations without changing into TCPIP thread using callbacks. See LOCK_TCPIP_CORE() and UNLOCK_TCPIP_CORE().

If disable tcpip core locking, TCP IP will perform tasks through context switching

*Default value:*
  - No (disabled)

**CONFIG_LWIP_TCPIP_CORE_LOCKING_INPUT**

Enable tcpip core locking input

*Found in: Component config > LWIP > CONFIG_LWIP_TCPIP_CORE_LOCKING*

When LWIP_TCPIP_CORE_LOCKING is enabled, this lets tcpip_input() grab the mutex for input packets as well, instead of allocating a message and passing it to tcpip_thread.

*Default value:*
  - No (disabled) if CONFIG_LWIP_TCPIP_CORE_LOCKING

**CONFIG_LWIP_CHECK_THREAD_SAFETY**

Checks that lwip API runs in expected context

*Found in: Component config > LWIP*

Enable to check that the project does not violate lwip thread safety. If enabled, all lwip functions that require thread awareness run an assertion to verify that the TCP/IP core functionality is either locked or accessed from the correct thread.

*Default value:*
  - No (disabled)

**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, 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.

*Default value:*

- Yes (enabled)

---

**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.

*Default value:*

- No (disabled)

---

**CONFIG_LWIP_EXTRA_IRAM_OPTIMIZATION**

Enable LWIP IRAM optimization for TCP part

*Found in: Component config > LWIP*

If this feature is enabled, some tcp part functions relating to RX/TX in LWIP will be put into IRAM, it can improve TCP throughput. On the other hand, it needs about 17KB IRAM for these optimizations.

*Default value:*

- No (disabled)

---

**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.

*Default value:*

- Yes (enabled)
CONFIG_LWIP_ND6

LWIP NDP6 Enable/Disable

*Found in: Component config > LWIP*

This option is used to disable the Network Discovery Protocol (NDP) if it is not required. Please use this option with caution, as the NDP is essential for IPv6 functionality within a local network.

**Default value:**
- Yes (enabled)

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.

**Range:**
- from 1 to 16

**Default value:**
- 10

CONFIG_LWIP_USE_ONLY_LWIP_SELECT

Support LWIP socket select() only (DEPRECATED)

*Found in: Component config > LWIP*

This option is deprecated. Do not use this option, use VFS_SUPPORT_SELECT instead.

**Default value:**
- No (disabled)

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.

**Default value:**
- No (disabled)

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.

**Default value:**
- Yes (enabled)
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.

**Default value:**
- Yes (enabled)

CONFIG_LWIP_SO_RCVBUF

Enable SO_RCVBUF option

*Found in: Component config > LWIP*

Enabling this option allows checking for available data on a netconn.

**Default value:**
- No (disabled)

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.

**Default value:**
- No (disabled)

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.

**Default value:**
- Yes (enabled)

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.

**Default value:**
- Yes (enabled)
**CONFIG_LWIP_IP4_REASSEMBLY**

Enable reassembly incoming fragmented IP4 packets

*Found in:* Component config > LWIP

Enabling this option allows reassembling incoming fragmented IP4 packets.

**Default value:**

- No (disabled)

**CONFIG_LWIP_IP6_REASSEMBLY**

Enable reassembly incoming fragmented IP6 packets

*Found in:* Component config > LWIP

Enabling this option allows reassembling incoming fragmented IP6 packets.

**Default value:**

- No (disabled)

**CONFIG_LWIP_IP_REASS_MAX_PBUFS**

The maximum amount of pbufs waiting to be reassembled

*Found in:* Component config > LWIP

Set the maximum amount of pbufs waiting to be reassembled.

**Range:**

- from 10 to 100

**Default value:**

- 10

**CONFIG_LWIP_IP_FORWARD**

Enable IP forwarding

*Found in:* Component config > LWIP

Enabling this option allows packets forwarding across multiple interfaces.

**Default value:**

- No (disabled)

**CONFIG_LWIP_IPV4_NAPT**

Enable NAT (new/experimental)

*Found in:* Component config > LWIP > CONFIG_LWIP_IP_FORWARD

Enabling this option allows Network Address and Port Translation.

**Default value:**

- No (disabled) if CONFIG_LWIP_IP_FORWARD

**CONFIG_LWIP_IPV4_NAPT_PORTMAP**

Enable NAT Port Mapping (new/experimental)

*Found in:* Component config > LWIP > CONFIG_LWIP_IP_FORWARD > CONFIG_LWIP_IPV4_NAPT

Enabling this option allows Port Forwarding or Port mapping.

**Default value:**

- Yes (enabled) if CONFIG_LWIP_IPV4_NAPT
CONFIG_LWIP_STATS

Enable LWIP statistics

*Found in: Component config > LWIP*

Enabling this option allows LWIP statistics

**Default value:**
- No (disabled)

CONFIG_LWIP_ESP_GRANTEDENTIAL_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.

**Default value:**
- Yes (enabled)

CONFIG_LWIP_GARP_TMR_INTERVAL

GARP timer interval(seconds)

*Found in: Component config > LWIP > CONFIG_LWIP_ESP_GRANTEDITIONAL_ARP*

Set the timer interval for gratuitous ARP. The default value is 60s

**Default value:**
- 60

CONFIG_LWIP_ESP_MLDV6_REPORT

Send mldv6 report periodically

*Found in: Component config > LWIP*

Enable this option allows to send mldv6 report periodically.

This option solve the issue that failed to receive multicast data. Some routers fail to forward multicast packets. To solve this problem, send multicast mldv6 report to routers regularly.

**Default value:**
- Yes (enabled)

CONFIG_LWIP_MLDV6_TMR_INTERVAL

mldv6 report timer interval(seconds)

*Found in: Component config > LWIP > CONFIG_LWIP_ESP_MLDV6_REPORT*

Set the timer interval for mldv6 report. The default value is 30s

**Default value:**
- 40
**CONFIG_LWIP_TCP/IP_RECEVBOX_SIZE**

TCP/IP task receive mail box size

*Found in: Component config > LWIP*

Set TCP/IP 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.

**Range:**
- from 6 to 1024 if `CONFIG_LWIP_WND_SCALE`

**Default value:**
- 32

**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.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- No (disabled)

**CONFIG_LWIP_DHCP_DISABLE_VENDOR_CLASS_ID**

DHCP: Disable Use of vendor class identification

*Found in: Component config > LWIP*

This option could be used to disable DHCP client vendor class identification. Set this option to “y” in order to exclude option 60 from DHCP packets.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- No (disabled)
CONFIG_LWIP_DHCP_OPTIONS_LEN
DHCP total option length

*Found in: Component config > LWIP*

Set total length of outgoing DHCP option msg. Generally bigger value means it can carry more options and values. If your code meets LWIP_ASSERT due to option value is too long. Please increase the LWIP_DHCP_OPTIONS_LEN value.

**Range:**
- from 68 to 255

**Default value:**
- 68

CONFIG_LWIP_NUM_NETIF_CLIENT_DATA
Number of clients store data in netif

*Found in: Component config > LWIP*

Number of clients that may store data in client_data member array of struct netif.

**Range:**
- from 0 to 256

**Default value:**
- 0

CONFIG_LWIP_DHCP_COARSE_TIMER_SECS
DHCP coarse timer interval(s)

*Found in: Component config > LWIP*

Set DHCP coarse interval in seconds. A higher value will be less precise but cost less power consumption.

**Range:**
- from 1 to 10

**Default value:**
- 1

DHCP server Contains:
- CONFIG_LWIP_DHCP

CONFIG_LWIP_DHCP
DHCPs: Enable IPv4 Dynamic Host Configuration Protocol Server (DHCPs)

*Found in: Component config > LWIP > DHCP server*

Enabling this option allows the device to run the DHCP server (to dynamically assign IPv4 addresses to clients).

**Default value:**
- Yes (enabled)

CONFIG_LWIP_DHCP_LEASE_UNIT
Multiplier for lease time, in seconds

*Found in: Component config > LWIP > DHCP server > CONFIG_LWIP_DHCP*

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.
Chapter 2. API Reference

**Range:**
- from 1 to 3600

**Default value:**
- 60

**CONFIG_LWIP_DHCPS_MAX_STATION_NUM**

Maximum number of stations

*Found in: Component config > LWIP > DHCP server > CONFIG_LWIP_DHCPS*

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 its address pool, without notification.

**Range:**
- from 1 to 64

**Default value:**
- 8

**CONFIG_LWIP_DHCPS_STATIC_ENTRIES**

Enable ARP static entries

*Found in: Component config > LWIP > DHCP server > CONFIG_LWIP_DHCPS*

Enabling this option allows DHCP server to support temporary static ARP entries for DHCP Client. This will help the DHCP server to send the DHCP OFFER and DHCP ACK using IP unicast.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- No (disabled)

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.)

**Range:**
- from 1 to 100 if **CONFIG_LWIP_AUTOIP**
Default value:
  • 2 if `CONFIG_LWIP_AUTOIP`

**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.

*Range:*
  • from 1 to 100 if `CONFIG_LWIP_AUTOIP`

*Default value:*
  • 9 if `CONFIG_LWIP_AUTOIP`

**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.

*Range:*
  • from 5 to 120 if `CONFIG_LWIP_AUTOIP`

*Default value:*
  • 20 if `CONFIG_LWIP_AUTOIP`

**CONFIG_LWIP_IPV4**

Enable IPv4

*Found in: Component config > LWIP*

Enable IPv4 stack. If you want to use IPv6 only TCP/IP stack, disable this.

*Default value:*
  • Yes (enabled)

**CONFIG_LWIP_IPV6**

Enable IPv6

*Found in: Component config > LWIP*

Enable IPv6 function. If not use IPv6 function, set this option to n. If disabling LWIP_IPV6 then some other components (coap and asio) will no longer be available.

*Default value:*
  • Yes (enabled)

**CONFIG_LWIP_IPV6_AUTOCONFIG**

Enable IPv6 stateless address autoconfiguration (SLAAC)

*Found in: Component config > LWIP > `CONFIG_LWIP_IPV6`*

Enabling this option allows the devices to IPv6 stateless address autoconfiguration (SLAAC).

See RFC 4862.

*Default value:*
  • No (disabled)
Chapter 2. API Reference

**CONFIG_LWIP_IPV6_NUM_ADDRESSES**

Number of IPv6 addresses on each network interface

*Found in: Component config > LWIP > CONFIG_LWIP_IPV6*

The maximum number of IPv6 addresses on each interface. Any additional addresses will be discarded.

**Default value:**

- 3

**CONFIG_LWIP_IPV6_FORWARD**

Enable IPv6 forwarding between interfaces

*Found in: Component config > LWIP > CONFIG_LWIP_IPV6*

Forwarding IPv6 packets between interfaces is only required when acting as a router.

**Default value:**

- No (disabled)

**CONFIG_LWIP_IPV6_RDNSS_MAX_DNS_SERVERS**

Use IPv6 Router Advertisement Recursive DNS Server Option

*Found in: Component config > LWIP*

Use IPv6 Router Advertisement Recursive DNS Server Option (as per RFC 6106) to copy a defined maximum number of DNS servers to the DNS module. Set this option to a number of desired DNS servers advertised in the RA protocol. This feature is disabled when set to 0.

**Default value:**

- 0 if `CONFIG_LWIP_IPV6_AUTOCONFIG`

**CONFIG_LWIP_IPV6_DHCP6**

Enable DHCPv6 stateless address autoconfiguration

*Found in: Component config > LWIP*

Enable DHCPv6 for IPv6 stateless address autoconfiguration. Note that the dhcpv6 client has to be started using `dhcp6_enable_stateless(netif)`. Note that the stateful address autoconfiguration is not supported.

**Default value:**

- No (disabled) if `CONFIG_LWIP_IPV6_AUTOCONFIG`

**CONFIG_LWIP_NETIF_STATUS_CALLBACK**

Enable status callback for network interfaces

*Found in: Component config > LWIP*

Enable callbacks when the network interface is up/down and addresses are changed.

**Default value:**

- No (disabled)
Chapter 2. API Reference

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. Disabling this option disables support of loopback interface in lwIP

Default value:
• Yes (enabled)

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.

Range:
• from 0 to 16

Default value:
• 8

TCP Contains:
• CONFIG_LWIP_TCP_WND_DEFAULT
• CONFIG_LWIP_TCP_SND_BUF_DEFAULT
• CONFIG_LWIP_TCP_RECVMBOX_SIZE
• CONFIG_LWIP_TCP_RTO_TIME
• CONFIG_LWIP_MAX_ACTIVE_TCP
• CONFIG_LWIP_TCP_FIN_WAIT_TIMEOUT
• CONFIG_LWIP_MAX_LISTENING_TCP
• CONFIG_LWIP_TCP_MAXRTX
• CONFIG_LWIP_TCP_SYNMAXRTX
• CONFIG_LWIP_TCP_MSL
• CONFIG_LWIP_TCP_MSS
• CONFIG_LWIP_TCP_OVERSIZE
• CONFIG_LWIP_TCP_QUEUE_OOSEQ
• CONFIG_LWIP_TCP_WND_SCALE
• CONFIG_LWIP_TCP_HIGH_SPEED_RETRANSMISSION
• CONFIG_LWIP_TCP_TMR_INTERVAL

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.

Range:
• from 1 to 1024
Default value:
• 16

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.

Range:
• from 1 to 1024
Default value:
• 16

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.

Default value:
• Yes (enabled)

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.

Range:
• from 3 to 12
Default value:
• 12

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.

Range:
• from 3 to 12
Default value:
• 12
**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

*Range:*
  - from 536 to 1460

*Default value:*
  - 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.

*Default value:*
  - 250

**CONFIG_LWIP_TCP_MSL**

Maximum segment lifetime (MSL)

*Found in: Component config > LWIP > TCP*

Set maximum segment lifetime in milliseconds.

*Default value:*
  - 60000

**CONFIG_LWIP_TCP_FIN_WAIT_TIMEOUT**

Maximum FIN segment lifetime

*Found in: Component config > LWIP > TCP*

Set maximum segment lifetime in milliseconds.

*Default value:*
  - 20000

**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.

*Range:*
  - from 2440 to 1024000 if CONFIG_LWIP_WND_SCALE

*Default value:*
• 5760

**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.

*Range:*
  • from 2440 to 1024000 if `CONFIG_LWIP_WND_SCALE`

*Default value:*
  • 5760

**CONFIG_LWIP_TCP_RECVMBOX_SIZE**

Default TCP receive mailbox size

*Found in: Component config > LWIP > TCP*

Set TCP receive mailbox size. Generally bigger value means higher throughput but more memory. The recommended value is: \( \text{LWIP_TCP_WND_DEFAULT/\text{TCP_MSS} + 2} \), e.g. if \( \text{LWIP_TCP_WND_DEFAULT}=14360 \), \( \text{TCP_MSS}=1436 \), then the recommended receive mailbox size is \( (14360/1436 + 2) = 12 \).

TCP receive mailbox is a per socket mailbox, when the application receives packets from TCP socket, LWIP core firstly posts the packets to TCP receive mailbox and the application then fetches the packets from mailbox. It means LWIP can caches maximum LWIP_TCP_RECVMBOX_SIZE packets for each TCP socket, so the maximum possible cached TCP packets for all TCP sockets is LWIP_TCP_RECVMBOX_SIZE multiples the maximum TCP socket number. In other words, the bigger LWIP_TCP_RECVMBOX_SIZE means more memory. On the other hand, if the receive mailbox is too small, the mailbox may be full. If the mailbox is full, the LWIP drops the packets. So generally we need to make sure the TCP receive mailbox is big enough to avoid packet drop between LWIP core and application.

*Range:*
  • from 6 to 1024 if `CONFIG_LWIP_WND_SCALE`

*Default value:*
  • 6

**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.

*Default value:*
  • Yes (enabled)
CONFIG_LWIP_TCP_OOSEQ_TIMEOUT

Timeout for each pbuf queued in TCP OOSEQ, in RTOs.

*Found in: Component config > LWIP > TCP > CONFIG_LWIP_TCP_QUEUE_OOSEQ*

The timeout value is TCP_OOSEQ_TIMEOUT * RTO.

**Range:**
- from 1 to 30

**Default value:**
- 6

CONFIG_LWIP_TCP_OOSEQ_MAX_PBUFS

The maximum number of pbufs queued on OOSEQ per pcb

*Found in: Component config > LWIP > TCP > CONFIG_LWIP_TCP_QUEUE_OOSEQ*

If LWIP_TCP_OOSEQ_MAX_PBUFS = 0, TCP will not control the number of OOSEQ pbufs.

In a poor network environment, many out-of-order tcp pbufs will be received. These out-of-order pbufs will be cached in the TCP out-of-order queue which will cause Wi-Fi/Ethernet fail to release RX buffer in time. It is possible that all RX buffers for MAC layer are used by OOSEQ.

Control the number of out-of-order pbufs to ensure that the MAC layer has enough RX buffer to receive packets.

In the Wi-Fi scenario, recommended OOSEQ PBUFS Range: 0 <= TCP_OOSEQ_MAX_PBUFS <= CONFIG_ESP_WIFI_DYNAMIC_RX_BUFFER_NUM/(MAX_TCP_NUMBER + 1)

In the Ethernet scenario, recommended Ethernet OOSEQ PBUFS Range: 0 <= TCP_OOSEQ_MAX_PBUFS <= CONFIG_ETH_DMA_RX_BUFFER_NUM/(MAX_TCP_NUMBER + 1)

Within the recommended value range, the larger the value, the better the performance.

MAX_TCP_NUMBER represent Maximum number of TCP connections in Wi-Fi(STA+SoftAP) and Ethernet scenario.

**Range:**
- from 0 to 12

**Default value:**
- 4

CONFIG_LWIP_TCP_SACK_OUT

Support sending selective acknowledgements

*Found in: Component config > LWIP > TCP > CONFIG_LWIP_TCP_QUEUE_OOSEQ*

TCP will support sending selective acknowledgements (SACKs).

**Default value:**
- No (disabled)

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 (CONFIG_LWIP_TCP_OVERSIZE_MSS)
- 25% MSS (CONFIG_LWIP_TCP_OVERSIZE_QUARTER_MSS)
- Disabled (CONFIG_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.

**Default value:**

- No (disabled) if `CONFIG_SPIRAM.TryAllocate_WIFI_LWIP`

**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.

**Range:**

- from 0 to 14 if `CONFIG_LWIP_WND_SCALE`

**Default value:**

- 0 if `CONFIG_LWIP_WND_SCALE`

**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.

**Default value:**

- 1500

**UDP**

Contains:

- `CONFIG_LWIP_UDP_RECVMBOX_SIZE`
- `CONFIG_LWIP_MAX_UDP_PCBS`

**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.

**Range:**

- from 1 to 1024

**Default value:**
• 16

**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_RECVMBOX_SIZE packets for each UDP socket, so the maximum possible cached UDP packets for all UDP sockets is UDP_RECVMBOX_SIZE multiples 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.

**Range:**
- from 6 to 64

**Default value:**
- 6

**Checksums**

Contains:

- **CONFIG_LWIP_CHECKSUM_CHECK_ICMP**
- **CONFIG_LWIP_CHECKSUM_CHECK_IP**
- **CONFIG_LWIP_CHECKSUM_CHECK_UDP**

**CONFIG_LWIP_CHECKSUM_CHECK_IP**

Enable LWIP IP checksums

*Found in: Component config > LWIP > Checksums*

Enable checksum checking for received IP messages

**Default value:**
- No (disabled)

**CONFIG_LWIP_CHECKSUM_CHECK_UDP**

Enable LWIP UDP checksums

*Found in: Component config > LWIP > Checksums*

Enable checksum checking for received UDP messages

**Default value:**
- No (disabled)

**CONFIG_LWIP_CHECKSUM_CHECK_ICMP**

Enable LWIP ICMP checksums

*Found in: Component config > LWIP > Checksums*

Enable checksum checking for received ICMP messages

**Default value:**
- Yes (enabled)
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.

**Range:**
- from 2048 to 65536

**Default value:**
- 3072

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 (CONFIG_LWIP_TCPIP_TASK_AFFINITY_NO_AFFINITY)
- CPU0 (CONFIG_LWIP_TCPIP_TASK_AFFINITY_CPU0)
- CPU1 (CONFIG_LWIP_TCPIP_TASK_AFFINITY_CPU1)

CONFIG_LWIP_PPP_SUPPORT

Enable PPP support

*Found in: Component config > LWIP*

Enable PPP stack. Now only PPP over serial is possible.

**Default value:**
- No (disabled)

**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.

**Default value:**
- Yes (enabled) if CONFIG_LWIP_PPP_SUPPORT & CONFIG_LWIP_IPV6

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_NUM_NEIGHBORS
Max number of entries in IPv6 neighbor cache

*Found in: Component config > LWIP*

Config max number of entries in IPv6 neighbor cache

**Range:**
- from 3 to 10

**Default value:**
- 5

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.

**Default value:**
- No (disabled) if CONFIG_LWIP_PPP_SUPPORT

CONFIG_LWIP_PPP_PAP_SUPPORT
Enable PAP support

*Found in: Component config > LWIP*

Enable Password Authentication Protocol (PAP) support

**Default value:**
- No (disabled) if CONFIG_LWIP_PPP_SUPPORT

CONFIG_LWIP_PPP_CHAP_SUPPORT
Enable CHAP support

*Found in: Component config > LWIP*

Enable Challenge Handshake Authentication Protocol (CHAP) support

**Default value:**
- No (disabled) if CONFIG_LWIP_PPP_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

**Default value:**
- No (disabled) if CONFIG_LWIP_PPP_SUPPORT
CONFIG_LWIP_PPP_MPPE_SUPPORT
Enable MPPE support

*Found in: Component config > LWIP*
Enable Microsoft Point-to-Point Encryption (MPPE) support

**Default value:**
- No (disabled) if `CONFIG_LWIP_PPP_SUPPORT`

CONFIG_LWIP_ENABLE_LCP_ECHO
Enable LCP ECHO

*Found in: Component config > LWIP*
Enable LCP echo keepalive requests

**Default value:**
- No (disabled) if `CONFIG_LWIP_PPP_SUPPORT`

CONFIG_LWIP_LCP_ECHOINTERVAL
Echo interval (s)

*Found in: Component config > LWIP > CONFIG_LWIP_ENABLE_LCP_ECHO*
Interval in seconds between keepalive LCP echo requests, 0 to disable.

**Range:**
- from 0 to 1000000 if `CONFIG_LWIP_ENABLE_LCP_ECHO`

**Default value:**
- 3 if `CONFIG_LWIP_ENABLE_LCP_ECHO`

CONFIG_LWIP_LCP_MAXECHOFAILS
Maximum echo failures

*Found in: Component config > LWIP > CONFIG_LWIP_ENABLE_LCP_ECHO*
Number of consecutive unanswered echo requests before failure is indicated.

**Range:**
- from 0 to 100000 if `CONFIG_LWIP_ENABLE_LCP_ECHO`

**Default value:**
- 3 if `CONFIG_LWIP_ENABLE_LCP_ECHO`

CONFIG_LWIP_PPP_DEBUG_ON
Enable PPP debug log output

*Found in: Component config > LWIP*
Enable PPP debug log output

**Default value:**
- No (disabled) if `CONFIG_LWIP_PPP_SUPPORT`
CONFIG_LWIP_SLIP_SUPPORT
Enable SLIP support (new/experimental)

*Found in: Component config > LWIP*

Enable SLIP stack. Now only SLIP over serial is possible.
SLIP over serial support is experimental and unsupported.

**Default value:**
- No (disabled)

Contains:
- CONFIG_LWIP_SLIP_DEBUG_ON

CONFIG_LWIP_SLIP_DEBUG_ON
Enable SLIP debug log output

*Found in: Component config > LWIP > CONFIG_LWIP_SLIP_SUPPORT*

Enable SLIP debug log output

**Default value:**
- No (disabled) if CONFIG_LWIP_SLIP_SUPPORT

ICMP
Contains:
- CONFIG_LWIP_ICMP
- CONFIG_LWIP_BROADCAST_PING
- CONFIG_LWIP_MULTICAST_PING

CONFIG_LWIP_ICMP
ICMP: Enable ICMP

*Found in: Component config > LWIP > ICMP*

Enable ICMP module for check network stability

**Default value:**
- Yes (enabled)

CONFIG_LWIP_MULTICAST_PING
Respond to multicast pings

*Found in: Component config > LWIP > ICMP*

**Default value:**
- No (disabled)

CONFIG_LWIP_BROADCAST_PING
Respond to broadcast pings

*Found in: Component config > LWIP > ICMP*

**Default value:**
- No (disabled)
**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.

**Range:**
- from 1 to 1024

**Default value:**
- 16

**SNTP** Contains:

- **CONFIG_LWIP_SNTP_MAX_SERVERS**
- **CONFIG_LWIP_SNTP_UPDATE_DELAY**
- **CONFIG_LWIP_DHCP_GET_NTP_SRV**

**CONFIG_LWIP_SNTP_MAX_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.

**Range:**
- from 1 to 16

**Default value:**
- 1

**CONFIG_LWIP_DHCP_GET_NTP_SRV**

Request NTP servers from DHCP

Found in: **Component config > LWIP > SNTP**

If enabled, LWIP will add ‘NTP’ to Parameter-Request Option sent via DHCP-request. DHCP server might reply with an NTP server address in option 42. SNTP callback for such replies should be set accordingly (see `sntp_servermode_dhcp()` func.)

**Default value:**
- No (disabled)

**CONFIG_LWIP_DHCP_MAX_NTP_SERVERS**

Maximum number of NTP servers acquired via DHCP

Found in: **Component config > LWIP > SNTP > CONFIG_LWIP_DHCP_GET_NTP_SRV**

Set maximum number of NTP servers acquired via DHCP-offer. Should be less or equal to “Maximum number of NTP servers”, any extra servers would be just ignored.

**Range:**
- from 1 to 16 if **CONFIG_LWIP_DHCP_GET_NTP_SRV**

**Default value:**
- 1 if **CONFIG_LWIP_DHCP_GET_NTP_SRV**
**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).

**Range:**
- from 15000 to 4294967295

**Default value:**
- 3600000

**CONFIG_LWIP_BRIDGEIF_MAX_PORTS**

Maximum number of bridge ports

*Found in: Component config > LWIP*

Set maximum number of ports a bridge can consists of.

**Range:**
- from 1 to 63

**Default value:**
- 7

**CONFIG_LWIP_ESP_LWIP_ASSERT**

Enable LWIP ASSERT checks

*Found in: Component config > LWIP*

Enable this option keeps LWIP assertion checks enabled. It is recommended to keep this option enabled. If asserts are disabled for the entire project, they are also disabled for LWIP and this option is ignored.

**Hooks**  Contains:

- `CONFIG_LWIP_HOOK_ND6_GET_GW`
- `CONFIG_LWIP_HOOK_IP6_INPUT`
- `CONFIG_LWIP_HOOK_IP6_ROUTE`
- `CONFIG_LWIP_HOOK_IP6_SELECT_SRC_ADDR`
- `CONFIG_LWIP_HOOK_NETCONN_EXTERNAL_RESOLVE`
- `CONFIG_LWIP_HOOK_TCP_ISN`

**CONFIG_LWIP_HOOK_TCP_ISN**

TCP ISN Hook

*Found in: Component config > LWIP > Hooks*

Enables to define a TCP ISN hook to randomize initial sequence number in TCP connection. The default TCP ISN algorithm used in IDF (standardized in RFC 6528) produces ISN by combining an MD5 of the new TCP id and a stable secret with the current time. This is because the lwIP implementation (`tcp_next_iss`) is not very strong, as it does not take into consideration any platform specific entropy source.

Set to `LWIP_HOOK_TCP_ISN_CUSTOM` to provide custom implementation. Set to `LWIP_HOOK_TCP_ISN_NONE` to use lwIP implementation.

**Available options:**
Chapter 2. API Reference

- No hook declared (CONFIG_LWIP_HOOK_TCP_ISN_NONE)
- Default implementation (CONFIG_LWIP_HOOK_TCP_ISN_DEFAULT)
- Custom implementation (CONFIG_LWIP_HOOK_TCP_ISN_CUSTOM)

**CONFIG_LWIP_HOOK_IP6_ROUTE**

IPv6 route Hook

*Found in: Component config > LWIP > Hooks*

Enables custom IPv6 route hook. Setting this to “default” provides weak implementation stub that could be overwritten in application code. Setting this to “custom” provides hook’s declaration only and expects the application to implement it.

Available options:

- No hook declared (CONFIG_LWIP_HOOK_IP6_ROUTE_NONE)
- Default (weak) implementation (CONFIG_LWIP_HOOK_IP6_ROUTE_DEFAULT)
- Custom implementation (CONFIG_LWIP_HOOK_IP6_ROUTE_CUSTOM)

**CONFIG_LWIP_HOOK_ND6_GET_GW**

IPv6 get gateway Hook

*Found in: Component config > LWIP > Hooks*

Enables custom IPv6 route hook. Setting this to “default” provides weak implementation stub that could be overwritten in application code. Setting this to “custom” provides hook’s declaration only and expects the application to implement it.

Available options:

- No hook declared (CONFIG_LWIP_HOOK_ND6_GET_GW_NONE)
- Default (weak) implementation (CONFIG_LWIP_HOOK_ND6_GET_GW_DEFAULT)
- Custom implementation (CONFIG_LWIP_HOOK_ND6_GET_GW_CUSTOM)

**CONFIG_LWIP_HOOK_IP6_SELECT_SRC_ADDR**

IPv6 source address selection Hook

*Found in: Component config > LWIP > Hooks*

Enables custom IPv6 source address selection. Setting this to “default” provides weak implementation stub that could be overwritten in application code. Setting this to “custom” provides hook’s declaration only and expects the application to implement it.

Available options:

- No hook declared (CONFIG_LWIP_HOOK_IP6_SELECT_SRC_ADDR_NONE)
- Default (weak) implementation (CONFIG_LWIP_HOOK_IP6_SELECT_SRC_ADDR_DEFAULT)
- Custom implementation (CONFIG_LWIP_HOOK_IP6_SELECT_SRC_ADDR_CUSTOM)

**CONFIG_LWIP_HOOK_NETCONN_EXTERNAL_RESOLVE**

Netconn external resolve Hook

*Found in: Component config > LWIP > Hooks*
Enables custom DNS resolve hook. Setting this to “default” provides weak implementation stub that could be overwritten in application code. Setting this to “custom” provides hook’s declaration only and expects the application to implement it.

Available options:

- No hook declared (CONFIG_LWIP_HOOK_NETCONN_EXT_RESOLVE_NONE)
- Default (weak) implementation (CONFIG_LWIP_HOOK_NETCONN_EXT_RESOLVE_DEFAULT)
- Custom implementation (CONFIG_LWIP_HOOK_NETCONN_EXT_RESOLVE_CUSTOM)

**CONFIG_LWIP_HOOK_IP6_INPUT**

IPv6 packet input

*Found in: Component config > LWIP > Hooks*

Enables custom IPv6 packet input. Setting this to “default” provides weak implementation stub that could be overwritten in application code. Setting this to “custom” provides hook’s declaration only and expects the application to implement it.

Available options:

- No hook declared (CONFIG_LWIP_HOOK_IP6_INPUT_NONE)
- Default (weak) implementation (CONFIG_LWIP_HOOK_IP6_INPUT_DEFAULT)
- Custom implementation (CONFIG_LWIP_HOOK_IP6_INPUT_CUSTOM)

**CONFIG_LWIP_DEBUG**

Enable LWIP Debug

*Found in: Component config > LWIP*

Enabling this option allows different kinds of lwIP debug output.

All lwIP debug features increase the size of the final binary.

**Default value:**

- No (disabled)

Contains:

- CONFIG_LWIP_API_LIB_DEBUG
- CONFIG_LWIP_BRIDGEIF_FDB_DEBUG
- CONFIG_LWIP_BRIDGEIF_FW_DEBUG
- CONFIG_LWIP_BRIDGEIF_DEBUG
- CONFIG_LWIP_DHCP_DEBUG
- CONFIG_LWIP_DHCP_STATE_DEBUG
- CONFIG_LWIP_DNS_DEBUG
- CONFIG_LWIP_ETHARP_DEBUG
- CONFIG_LWIP_ICMP_DEBUG
- CONFIG_LWIP_ICMP6_DEBUG
- CONFIG_LWIP_IP_DEBUG
- CONFIG_LWIP_IP6_DEBUG
- CONFIG_LWIP_NAPT_DEBUG
- CONFIG_LWIP_NETIF_DEBUG
- CONFIG_LWIP_PBUF_DEBUG
- CONFIG_LWIP_SNTP_DEBUG
- CONFIG_LWIP_SOCKETS_DEBUG
- CONFIG_LWIP_TCP_DEBUG
- CONFIG_LWIP_UDP_DEBUG
- CONFIG_LWIP_DEBUG_ESP_LOG
Chapter 2. API Reference

**CONFIG_LWIP_DEBUG_ESP_LOG**

Route LWIP debugs through ESP_LOG interface

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

Enabling this option routes all enabled LWIP debugs through ESP_LOGD.

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_NETIF_DEBUG**

Enable netif debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_PBUF_DEBUG**

Enable pbuf debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_ETHARP_DEBUG**

Enable etharp debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_API_LIB_DEBUG**

Enable api lib debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_SOCKETS_DEBUG**

Enable socket debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_IP_DEBUG**

Enable IP debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`
**CONFIG_LWIP_ICMP_DEBUG**

Enable ICMP debug messages

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG` & `CONFIG_LWIP_ICMP`

**CONFIG_LWIP_DHCP_STATE_DEBUG**

Enable DHCP state tracking

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_DHCP_DEBUG**

Enable DHCP debug messages

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_IP6_DEBUG**

Enable IP6 debug messages

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_ICMP6_DEBUG**

Enable ICMP6 debug messages

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_TCP_DEBUG**

Enable TCP debug messages

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG`

**CONFIG_LWIP_UDP_DEBUG**

Enable UDP debug messages

*Found in:* Component config > LWIP > CONFIG_LWIP_DEBUG

**Default value:**

- No (disabled) if `CONFIG_LWIP_DEBUG`
CONFIG_LWIP_SNTP_DEBUG

Enable SNTP debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

CONFIG_LWIP_DNS_DEBUG

Enable DNS debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

CONFIG_LWIP_NAPT_DEBUG

Enable NAPT debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG` & `CONFIG_LWIP_IPV4_NAPT`

CONFIG_LWIP_BRIDGEIF_DEBUG

Enable bridge generic debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

CONFIG_LWIP_BRIDGEIF_FDB_DEBUG

Enable bridge FDB debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

CONFIG_LWIP_BRIDGEIF_FW_DEBUG

Enable bridge forwarding debug messages

*Found in: Component config > LWIP > CONFIG_LWIP_DEBUG*

**Default value:**
- No (disabled) if `CONFIG_LWIP_DEBUG`

mbedTLS  Contains:

- `CONFIG_MBEDTLS_SYMMETRIC_CONTENT_LEN`
- Certificate Bundle
- Certificates
- `CONFIG_MBEDTLS_CHACHA20_C`
- `CONFIG_MBEDTLS_DH_C`
- `CONFIG_MBEDTLS_ECP_C`
- `CONFIG_MBEDTLS_ECDH_C`
• CONFIGMBEDTLS_ECJPAKE_C
• CONFIGMBEDTLS_ECP_DP_BP256R1_ENABLED
• CONFIGMBEDTLS_ECP_DP_BP384R1_ENABLED
• CONFIGMBEDTLS_ECP_DP_BP512R1_ENABLED
• CONFIGMBEDTLS_CMAC_C
• CONFIGMBEDTLS_ECP_DP_CURVE25519_ENABLED
• CONFIGMBEDTLS_ECDSA_DETERMINISTIC
• CONFIGMBEDTLS_HARDWARE_ECDSA_VERIFY
• CONFIGMBEDTLS_HARDWARE_ECDSA_SIGN
• CONFIGMBEDTLS_ECP_FIXED_POINT_OPTIM
• CONFIGMBEDTLS_HARDWARE_AES
• CONFIGMBEDTLS_HARDWARE_ECC
• CONFIGMBEDTLS_ATCA_HW_ECDSA_SIGN
• CONFIGMBEDTLS_ATCA_HW_ECDSA_VERIFY
• CONFIGMBEDTLS_HARDWARE_MPI
• CONFIGMBEDTLS_HARDWARE_SHA
• CONFIGMBEDTLS_DEBUG
• CONFIGMBEDTLS_ECP_RESTARTABLE
• CONFIGMBEDTLS_HAVE_TIME
• CONFIGMBEDTLS_RIPEMD160_C
• CONFIGMBEDTLS_ECP_DP_SECP192K1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP192R1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP224K1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP224R1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP256K1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP256R1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP384R1_ENABLED
• CONFIGMBEDTLS_ECP_DP_SECP521R1_ENABLED
• CONFIGMBEDTLS_SHA512_C
• CONFIGMBEDTLS_THREADING_C
• CONFIGMBEDTLS_LARGE_KEY_SOFTWARE_MPI
• CONFIGMBEDTLS_HKDF_C
• mbedTLS v3.x related
• CONFIGMBEDTLS_MEM_ALLOC_MODE
• CONFIGMBEDTLS_ECP_NIST_OPTIM
• CONFIGMBEDTLS_POLY1305_C
• CONFIGMBEDTLS_SSL_ALPN
• CONFIGMBEDTLS_SSL_PROTO_DTLS
• CONFIGMBEDTLS_SSL_PROTO_GMTSSL1_1
• CONFIGMBEDTLS_SSL_PROTO_TLS1_2
• CONFIGMBEDTLS_SSL_RENEGOTIATION
• Symmetric Ciphers
• TLS Key Exchange Methods
• CONFIGMBEDTLS_SSL_MAX_CONTENT_LEN
• CONFIGMBEDTLS_TLS_MODE
• CONFIGMBEDTLS_CLIENT_SSL_SESSION_TICKETS
• CONFIGMBEDTLS_SERVER_SSL_SESSION_TICKETS
• CONFIGMBEDTLS_ROM_MD5
• CONFIGMBEDTLS_USE_CRYPTO_ROM_IMPL
• CONFIGMBEDTLS_DYNAMIC_BUFFER

CONFIGMBEDTLS_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_calloc_free() function
• Internal IRAM memory wherever applicable else internal DRAM

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.

(*) In case of ESP32-S2/ESP32-S3, hardware allows encryption of external SPIRAM contents provided hardware flash encryption feature is enabled. In that case, using external SPIRAM allocation strategy is also safe choice from security perspective.

Available options:

• Internal memory (CONFIG_MBEDTLS_INTERNAL_MEM_ALLOC)
• External SPIRAM (CONFIG_MBEDTLS_EXTERNAL_MEM_ALLOC)
• Default alloc mode (CONFIG_MBEDTLS_DEFAULT_MEM_ALLOC)
• Custom alloc mode (CONFIG_MBEDTLS_CUSTOM_MEM_ALLOC)
• Internal IRAM (CONFIG_MBEDTLS_IRAM_8BIT_MEM_ALLOC)

Allows to use IRAM memory region as 8bit accessible region.

TLS input and output buffers will be allocated in IRAM section which is 32bit aligned memory. Every unaligned (8bit or 16bit) access will result in an exception and incur penalty of certain clock cycles per unaligned read/write.

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.

*Default value:*

• Yes (enabled)

CONFIG_MBEDTLS_SSL_IN_CONTENT_LEN

TLS maximum incoming fragment length

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_ASYMMETRIC_CONTENT_LEN*
Chapter 2. API Reference

This defines maximum incoming fragment length, overriding default maximum content length (MBEDTLS_SSL_MAX_CONTENT_LEN).

Range:
- from 512 to 16384

Default value:
- 16384

**CONFIG_MBEDTLS_SSL_OUT_CONTENT_LEN**

TLS maximum outgoing fragment length

*Found in: Component config > mbedTLS > CONFIG_MBEDTLSASYMMETRIC_CONTENT_LEN*

This defines maximum outgoing fragment length, overriding default maximum content length (MBEDTLS_SSL_MAX_CONTENT_LEN).

Range:
- from 512 to 16384

Default value:
- 4096

**CONFIG_MBEDTLS_DYNAMIC_BUFFER**

Using dynamic TX/RX buffer

*Found in: Component config > mbedTLS*

Using dynamic TX/RX buffer. After enabling this option, mbedTLS will allocate TX buffer when need to send data and then free it if all data is sent, allocate RX buffer when need to receive data and then free it when all data is used or read by upper layer.

By default, when SSL is initialized, mbedTLS also allocate TX and RX buffer with the default value of “MBEDTLS_SSL_OUT_CONTENT_LEN” or “MBEDTLS_SSL_IN_CONTENT_LEN”, so to save more heap, users can set the options to be an appropriate value.

**CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA**

Free private key and DHM data after its usage

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_DYNAMIC_BUFFER*

Free private key and DHM data after its usage in handshake process.

The option will decrease heap cost when handshake, but also lead to problem:

Because all certificate, private key and DHM data are freed so users should register certificate and private key to ssl config object again.

Default value:
- No (disabled) if CONFIG_MBEDTLS_DYNAMIC_BUFFER

**CONFIG_MBEDTLS_DYNAMIC_FREE_CA_CERT**

Free SSL CA certificate after its usage

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_DYNAMIC_BUFFER > CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA*

Free CA certificate after its usage in the handshake process. This option will decrease the heap footprint for the TLS handshake, but may lead to a problem: If the respective ssl object needs to perform the TLS handshake again, the CA certificate should once again be registered to the ssl object.

Default value:
- Yes (enabled) if CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA
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.

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_DEBUG_LEVEL
Set mbedTLS debugging level

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_DEBUG*

Set mbedTLS debugging level

**Available options:**
- Warning (CONFIG_MBEDTLS_DEBUG_LEVEL_WARN)
- Info (CONFIG_MBEDTLS_DEBUG_LEVEL_INFO)
- Debug (CONFIG_MBEDTLS_DEBUG_LEVEL_DEBUG)
- Verbose (CONFIG_MBEDTLS_DEBUG_LEVEL_VERBOSE)

**mbedTLS v3.x related**

Contains:
- DTLS-based configurations
- CONFIG_MBEDTLS_PKCS7_C
- CONFIG_MBEDTLS_SSL_CONTEXT.Serialization
- CONFIG_MBEDTLS_X509_TRUSTED_CERT_CALLBACK
- CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE
- CONFIG_MBEDTLS_SSL_CID_PADDING_GRANULARITY
- CONFIG_MBEDTLS_SSL_PROTO_TLS1_3
- CONFIG_MBEDTLS_ECDH_LEGACY_CONTEXT
- CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH

CONFIG_MBEDTLS_SSL_PROTO_TLS1_3
Support TLS 1.3 protocol

*Found in: Component config > mbedTLS > mbedTLS v3.x related*

**TLS 1.3 related configurations**

Contains:
- CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_EPHEMERAL
- CONFIG_MBEDTLS_SSL_TLS1_3_COMPATIBILITY_MODE
- CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK_EPHEMERAL
- CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK

CONFIG_MBEDTLS_SSL_TLS1_3_COMPATIBILITY_MODE
TLS 1.3 middlebox compatibility mode

*Found in: Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations*
**Default value:**

- Yes (enabled) if `CONFIG_MBEDTLS_SSL_PROTO_TLS1_3`

**CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK**

TLS 1.3 PSK key exchange mode

*Found in:* Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

**Default value:**

- Yes (enabled) if `CONFIG_MBEDTLS_SSL_PROTO_TLS1_3`

**CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_EPHEMERAL**

TLS 1.3 ephemeral key exchange mode

*Found in:* Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

**Default value:**

- Yes (enabled) if `CONFIG_MBEDTLS_SSL_PROTO_TLS1_3`

**CONFIG_MBEDTLS_SSL_TLS1_3_KEXM_PSK_EPHEMERAL**

TLS 1.3 PSK ephemeral key exchange mode

*Found in:* Component config > mbedTLS > mbedTLS v3.x related > CONFIG_MBEDTLS_SSL_PROTO_TLS1_3 > TLS 1.3 related configurations

**Default value:**

- Yes (enabled) if `CONFIG_MBEDTLS_SSL_PROTO_TLS1_3`

**CONFIG_MBEDTLS_SSL_VARIABLE_BUFFER_LENGTH**

Variable SSL buffer length

*Found in:* `Component config` > `mbedTLS` > `mbedTLS v3.x related`

This enables the SSL buffer to be resized automatically based on the negotiated maximum fragment length in each direction.

**Default value:**

- No (disabled)

**CONFIG_MBEDTLS_ECDH_LEGACY_CONTEXT**

Use a backward compatible ECDH context (Experimental)

*Found in:* `Component config` > `mbedTLS` > `mbedTLS v3.x related`

Use the legacy ECDH context format. Define this option only if you enable `MBEDTLS_ECP_RESTARTABLE` or if you want to access ECDH context fields directly.

**Default value:**

- No (disabled) if `CONFIG_MBEDTLS_ECDH_C` && `CONFIG_MBEDTLS_ECP_RESTARTABLE`
CONFIG_MBEDTLS_X509_TRUSTED_CERT_CALLBACK
Enable trusted certificate callbacks

*Found in: Component config > mbedTLS > mbedTLS v3.x related*

Enables users to configure the set of trusted certificates through a callback instead of a linked list.
See mbedTLS documentation for required API and more details.

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_SSL_CONTEXT.Serialization
Enable serialization of the TLS context structures

*Found in: Component config > mbedTLS > mbedTLS v3.x related*

Enable serialization of the TLS context structures. This is a local optimization in handling a single, potentially long-lived connection.

See mbedTLS documentation for required API and more details. Disabling this option will save some code size.

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_SSL_KEEP_PEER_CERTIFICATE
Keep peer certificate after handshake completion

*Found in: Component config > mbedTLS > mbedTLS v3.x related*

Keep the peer’s certificate after completion of the handshake. Disabling this option will save about 4kB of heap and some code size.

See mbedTLS documentation for required API and more details.

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_PKCS7_C
Enable PKCS #7

*Found in: Component config > mbedTLS > mbedTLS v3.x related*

Enable PKCS #7 core for using PKCS #7-formatted signatures.

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_SSL_CID_PADDING_GRANULARITY
Record plaintext padding

*Found in: Component config > mbedTLS > mbedTLS v3.x related*

Controls the use of record plaintext padding in TLS 1.3 and when using the Connection ID extension in DTLS 1.2.

The padding will always be chosen so that the length of the padded plaintext is a multiple of the value of this option.

**Notes:** A value of 1 means that no padding will be used for outgoing records. On systems lacking division instructions, a power of two should be preferred.
Range:
• from 0 to 32 if \texttt{CONFIG_MBEDTLS_SSL_PROTO_TLS1_3} \texttt{||} \texttt{CONFIG_MBEDTLS_SSL_PROTO_DTLS} \texttt{&&} \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID}

Default value:
• 16 if \texttt{CONFIG_MBEDTLS_SSL_PROTO_TLS1_3} \texttt{||} \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID}

\textbf{DTLS-based configurations} Contains:
• \texttt{CONFIG_MBEDTLS_SSL_DTLS_SRTP}
• \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID}

\textbf{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID}
Support for the DTLS Connection ID extension

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations
Enable support for the DTLS Connection ID extension which allows to identify DTLS connections across changes in the underlying transport.

Default value:
• No (disabled) if \texttt{CONFIG_MBEDTLS_SSL_PROTO_DTLS}

\textbf{CONFIG_MBEDTLS_SSL_CID_IN_LEN_MAX}
Maximum length of CIDs used for incoming DTLS messages

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations > \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID}
Maximum length of CIDs used for incoming DTLS messages

Range:
• from 0 to 32 if \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID} \texttt{&&} \texttt{CONFIG_MBEDTLS_SSL_PROTO_DTLS}

Default value:
• 32 if \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID} \texttt{&&} \texttt{CONFIG_MBEDTLS_SSL_PROTO_DTLS}

\textbf{CONFIG_MBEDTLS_SSL_CID_OUT_LEN_MAX}
Maximum length of CIDs used for outgoing DTLS messages

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations > \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID}
Maximum length of CIDs used for outgoing DTLS messages

Range:
• from 0 to 32 if \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID} \texttt{&&} \texttt{CONFIG_MBEDTLS_SSL_PROTO_DTLS}

Default value:
• 32 if \texttt{CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID} \texttt{&&} \texttt{CONFIG_MBEDTLS_SSL_PROTO_DTLS}

\textbf{CONFIG_MBEDTLS_SSL_DTLS_SRTP}
Enable support for negotiation of DTLS-SRTP (RFC 5764)

Found in: Component config > mbedTLS > mbedTLS v3.x related > DTLS-based configurations
Enable support for negotiation of DTLS-SRTP (RFC 5764) through the use_srtp extension.
See mbedTLS documentation for required API and more details. Disabling this option will save some code size.

**Default value:**
- No (disabled) if `CONFIG_MBEDTLS_SSL_PROTO_DTLS`

**Certificate Bundle** Contains:
- `CONFIG_MBEDTLS_CERTIFICATE_BUNDLE`

**CONFIG_MBEDTLS_CERTIFICATE_BUNDLE**

Enable trusted root certificate bundle

*Found in: Component config > mbedTLS > Certificate Bundle*

Enable support for large number of default root certificates

When enabled this option allows user to store default as well as customer specific root certificates in compressed format rather than storing full certificate. For the root certificates the public key and the subject name will be stored.

**Default value:**
- Yes (enabled)

**CONFIG_MBEDTLS_DEFAULT_CERTIFICATE_BUNDLE**

Default certificate bundle options

*Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE*

Available options:

- Use the full default certificate bundle (CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_FULL)
- Use only the most common certificates from the default bundles (CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_CMN)
  Use only the most common certificates from the default bundles, reducing the size with 50%, while still having around 99% coverage.
- Do not use the default certificate bundle (CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_NONE)

**CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE**

Add custom certificates to the default bundle

*Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE*

**Default value:**
- No (disabled)

**CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE_PATH**

Custom certificate bundle path

*Found in: Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE > CONFIG_MBEDTLS_CUSTOM_CERTIFICATE_BUNDLE***
CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEPRECATED_LIST

Add deprecated root certificates

*Found in:* Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE

Include the deprecated list of root certificates in the bundle. This list gets updated when a certificate is removed from the Mozilla’s NSS root certificate store. This config can be enabled if you would like to ensure that none of the certificates that were deployed in the product are affected because of the update to bundle. In turn, enabling this config keeps expired, retracted certificates in the bundle and it may pose a security risk.

- Deprecated cert list may grow based on sync with upstream bundle
- Deprecated certs would be removed in ESP-IDF (next) major release

CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_MAX_CERTS

Maximum no of certificates allowed in certificate bundle

*Found in:* Component config > mbedTLS > Certificate Bundle > CONFIG_MBEDTLS_CERTIFICATE_BUNDLE

*Default value:*

- 200

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.

*Default value:*

- No (disabled)

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.

*Default value:*

- No (disabled)

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_AES_USE_INTERRUPT

Use interrupt for long AES operations

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_AES*

Use an interrupt to coordinate long AES operations.

This allows other code to run on the CPU while an AES operation is pending. Otherwise the CPU busy-waits.

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_AES_INTERRUPT_LEVEL

AES hardware interrupt level

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_AES > CONFIG_MBEDTLS_AES_USE_INTERRUPT*

This config helps to set the interrupt priority level for the AES peripheral. Value 0 (default) means that there is no preference regarding the interrupt priority level and any level from 1 to 3 can be selected (based on the availability). Note: Higher value indicates high interrupt priority.

**Range:**
- from 0 to 3

**Default value:**
- 0

CONFIG_MBEDTLS_HARDWARE_GCM

Enable partially hardware accelerated GCM

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_AES*

Enable partially hardware accelerated GCM. GHASH calculation is still done in software.

If MBEDTLS_HARDWARE_GCM is disabled and MBEDTLS_HARDWARE_AES is enabled then mbedTLS will still use the hardware accelerated AES block operation, but on a single block at a time.

**Default value:**
- Yes (enabled)

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 SOC_RSA_MAX_BIT_LEN bit results.

These operations are used by RSA.

CONFIG_MBEDTLS_MPI_USE_INTERRUPT

Use interrupt for MPI exp-mod operations

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_MPI*

Use an interrupt to coordinate long MPI operations.

This allows other code to run on the CPU while an MPI operation is pending. Otherwise the CPU busy-waits.
Default value:
  • Yes (enabled)

**CONFIG_MBEDTLS_MPIInterrupt_LEVEL**

MPI hardware interrupt level

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_MPI > CONFIG_MBEDTLS_MPI_USE_INTERRUPT*

This config helps to set the interrupt priority level for the MPI peripheral. Value 0 (default) means that there is no preference regarding the interrupt priority level and any level from 1 to 3 can be selected (based on the availability). Note: Higher value indicates high interrupt priority.

**Range:**
  • from 0 to 3

**Default value:**
  • 0

**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, on the ESP32 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_HARDWARE_ECC**

Enable hardware ECC acceleration

*Found in: Component config > mbedTLS*

Enable hardware accelerated ECC point multiplication and point verification for points on curve SECP192R1 and SECP256R1 in mbedTLS

**Default value:**
  • Yes (enabled) if SOC_ECC_SUPPORTED

**CONFIG_MBEDTLS_ECC_OTHER_CURVES_SOFT_FALLBACK**

Fallback to software implementation for curves not supported in hardware

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HARDWARE_ECC*

Fallback to software implementation of ECC point multiplication and point verification for curves not supported in hardware.

**Default value:**
  • Yes (enabled) if CONFIG_MBEDTLS_HARDWARE_ECC
**CONFIG_MBEDTLS_ROM_MD5**

Use MD5 implementation in ROM

*Found in: Component config > mbedTLS*

Use ROM MD5 in mbedTLS.

*Default value:*

*Yes (enabled)*

**CONFIG_MBEDTLS_HARDWARE_ECDSA_SIGN**

Enable ECDSA signing using on-chip ECDSA peripheral

*Found in: Component config > mbedTLS*

Enable hardware accelerated ECDSA peripheral to sign data on curve SECP192R1 and SECP256R1 in mbedTLS.

Note that for signing, the private key has to be burnt in an efuse key block with key purpose set to ECDSA_KEY. If no key is burnt, it will report an error.

The key should be burnt in little endian format. espfuse.py utility handles it internally but care needs to be taken while burning using esp_efuse APIs.

*Default value:*

*No (disabled) if SOC_ECDSA_SUPPORTED*

**CONFIG_MBEDTLS_HARDWARE_ECDSA_VERIFY**

Enable ECDSA signature verification using on-chip ECDSA peripheral

*Found in: Component config > mbedTLS*

Enable hardware accelerated ECDSA peripheral to verify signature on curve SECP192R1 and SECP256R1 in mbedTLS.

*Default value:*

*Yes (enabled) if SOC_ECDSA_SUPPORTED*

**CONFIG_MBEDTLS_ATCA_HW_ECDSA_SIGN**

Enable hardware ECDSA sign acceleration when using ATECC608A

*Found in: Component config > mbedTLS*

This option enables hardware acceleration for ECDSA sign function, only when using ATECC608A cryptoauth chip (integrated with ESP32-WROOM-32SE)

*Default value:*

*No (disabled)*

**CONFIG_MBEDTLS_ATCA_HW_ECDSA_VERIFY**

Enable hardware ECDSA verify acceleration when using ATECC608A

*Found in: Component config > mbedTLS*

This option enables hardware acceleration for ECDSA sign function, only when using ATECC608A cryptoauth chip (integrated with ESP32-WROOM-32SE)

*Default value:*

*No (disabled)*
CONFIG_MBEDTLS_HAVE_TIME

Enable mbedTLS time support

*Found in: Component config > mbedTLS*

Enable use of time.h functions (time() and gmtime()) by mbedTLS. This option doesn’t require the system time to be correct, but enables functionality that requires relative timekeeping - for example periodic expiry of TLS session tickets or session cache entries. Disabling this option will save some firmware size, particularly if the rest of the firmware doesn’t call any standard timekeeping functions.

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_PLATFORM_TIME_ALT

Enable mbedTLS time support: platform-specific

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HAVE_TIME*

Enabling this config will provide users with a function “mbedtls_platform_set_time()” that allows to set an alternative time function pointer.

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_HAVE_TIME_DATE

Enable mbedTLS certificate expiry check

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_HAVE_TIME*

Enables X.509 certificate expiry checks in mbedTLS. If this option is disabled (default) then X.509 certificate “valid from” and “valid to” timestamp fields are ignored. If this option is enabled, these fields are compared with the current system date and time. The time is retrieved using the standard time() and gmtime() functions. If the certificate is not valid for the current system time then verification will fail with code MBEDTLS_X509_BADCERT_FUTURE or MBEDTLS_X509_BADCERT_EXPIRED.

Enabling this option requires adding functionality in the firmware to set the system clock to a valid timestamp before using TLS. The recommended way to do this is via ESP-IDF’s SNTP functionality, but any method can be used. In the case where only a small number of certificates are trusted by the device, please carefully consider the tradeoffs of enabling this option. There may be undesired consequences, for example if all trusted certificates expire while the device is offline and a TLS connection is required to update. Or if an issue with the SNTP server means that the system time is invalid for an extended period after a reset.

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_ECDSA_DETERMINISTIC

Enable deterministic ECDSA

*Found in: Component config > mbedTLS*

Standard ECDSA is “fragile” in the sense that lack of entropy when signing may result in a compromise of the long-term signing key.

**Default value:**

Chapter 2. API Reference

• Yes (enabled)

**CONFIG_MBEDTLS_SHA512_C**

Enable the SHA-384 and SHA-512 cryptographic hash algorithms

*Found in: Component config > mbedTLS*

Enable MBEDTLS_SHA512_C adds support for SHA-384 and SHA-512.

*Default value:*

• Yes (enabled)

**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 (CONFIG_MBEDTLS_TLS_SERVER_AND_CLIENT)
• Server (CONFIG_MBEDTLS_TLS_SERVER_ONLY)
• Client (CONFIG_MBEDTLS_TLS_CLIENT_ONLY)
• None (CONFIG_MBEDTLS_TLS_DISABLED)

**TLS Key Exchange Methods**

Contains:

• CONFIG_MBEDTLS_KEY_EXCHANGE_DHE_RSA
• CONFIG_MBEDTLS_KEY_EXCHANGE_ECJPAKE
• CONFIG_MBEDTLS_PSK_MODES
• CONFIG_MBEDTLS_KEY_EXCHANGE_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.

*Default value:*

• No (disabled)

**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.

*Default value:*

• No (disabled) if CONFIG_MBEDTLS_PSK_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.

**Default value:**
- Yes (enabled) if `CONFIG_MBEDTLS_PSK_MODES` && `CONFIG_MBEDTLS_DHM_C`

**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.

**Default value:**
- Yes (enabled) if `CONFIG_MBEDTLS_PSK_MODES` && `CONFIG_MBEDTLS_ECDH_C`

**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.

**Default value:**
- Yes (enabled) if `CONFIG_MBEDTLS_PSK_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-

**Default value:**
- Yes (enabled)

**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-

**Default value:**
- Yes (enabled) if `CONFIG_MBEDTLS_DHM_C`

**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.

**Default value:**
- Yes (enabled)

**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-

**Default value:**
- Yes (enabled)

**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-

**Default value:**
- Yes (enabled)

**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-

**Default value:**
- Yes (enabled)

**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-

**Default value:**
- Yes (enabled)

**CONFIG_MBEDTLS_KEY_EXCHANGE_ECJPAKE**

Enable ECJPAKE based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support ciphersuites with prefix TLS-ECJPAKE-WITH-

**Default value:**
- No (disabled) if CONFIG_MBEDTLS_ECJPAKE_C && CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED
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.

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_SSL_PROTO_TLS1_2
Support TLS 1.2 protocol

*Found in: Component config > mbedTLS*

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_SSL_PROTO_GMTSSL1_1
Support GM/T SSL 1.1 protocol

*Found in: Component config > mbedTLS*

Provisions for GM/T SSL 1.1 support

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_SSL_PROTO_DTLS
Support DTLS protocol (all versions)

*Found in: Component config > mbedTLS*

Requires TLS 1.2 to be enabled for DTLS 1.2

**Default value:**
- No (disabled)

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.

**Default value:**
- Yes (enabled)

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.

**Default value:**
- Yes (enabled)
**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.

**Default value:**
- Yes (enabled)

**Symmetric Ciphers**

Contains:

- CONFIG_MBEDTLS_AES_C
- CONFIG_MBEDTLS_BLOWFISH_C
- CONFIG_MBEDTLS_CAMELLIA_C
- CONFIG_MBEDTLS_CCM_C
- CONFIG_MBEDTLS_DES_C
- CONFIG_MBEDTLS_GCM_C
- CONFIG_MBEDTLS_NIST_KW_C
- CONFIG_MBEDTLS_XTEA_C

**CONFIG_MBEDTLS_AES_C**

AES block cipher

*Found in: Component config > mbedTLS > Symmetric Ciphers*

**Default value:**
- Yes (enabled)

**CONFIG_MBEDTLS_CAMELLIA_C**

Camellia block cipher

*Found in: Component config > mbedTLS > Symmetric Ciphers*

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled)

**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.
Default value:
• No (disabled)

CONFIG_MBEDTLS_XTEA_C
XTEA block cipher

Found in: Component config > mbedTLS > Symmetric Ciphers
Enables the XTEA block cipher.

Default value:
• No (disabled)

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.

Default value:
• Yes (enabled)

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.

Default value:
• Yes (enabled)

CONFIG_MBEDTLS_NIST_KW_C
NIST key wrapping (KW) and KW padding (KWP)

Found in: Component config > mbedTLS > Symmetric Ciphers
Enable NIST key wrapping and key wrapping padding.

Default value:
• No (disabled)

CONFIG_MBEDTLS_RIPEMD160_C
Enable RIPEMD-160 hash algorithm

Found in: Component config > mbedTLS
Enable the RIPEMD-160 hash algorithm.

Default value:
• No (disabled)
Chapter 2. API Reference

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_PEMPARSE_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.

**Default value:**
- Yes (enabled)

`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.

**Default value:**
- Yes (enabled)

`CONFIG_MBEDTLS_X509_CRLPARSE_C`
X.509 CRL parsing

*Found in: Component config > mbedTLS > Certificates*


**Default value:**
- Yes (enabled)

`CONFIG_MBEDTLS_X509_CSRPARSE_C`
X.509 CSR parsing

*Found in: Component config > mbedTLS > Certificates*

Support for parsing X.509 Certificate Signing Requests

**Default value:**
- Yes (enabled)

`CONFIG_MBEDTLS_ECP_C`
Elliptic Curve Ciphers

*Found in: Component config > mbedTLS*

**Default value:**
- Yes (enabled)
**CONFIG_MBEDTLS_DH_C**

Diffie-Hellman-Merkle key exchange (DHM)

*Found in: Component config > mbedTLS*

Enable DHM. Needed to use DHE-xxx TLS ciphersuites.

Note that the security of Diffie-Hellman key exchanges depends on a suitable prime being used for the exchange. Please see detailed warning text about this in file `mbedtls/dhm.h` file.

**Default value:**
- No (disabled)

**CONFIG_MBEDTLS_ECDH_C**

Elliptic Curve Diffie-Hellman (ECDH)

*Found in: Component config > mbedTLS*

Enable ECDH. Needed to use ECDHE-xxx TLS ciphersuites.

**Default value:**
- Yes (enabled)

**CONFIG_MBEDTLS_ECDSA_C**

Elliptic Curve DSA

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_ECDH_C*

Enable ECDSA. Needed to use ECDSA-xxx TLS ciphersuites.

**Default value:**
- Yes (enabled)

**CONFIG_MBEDTLS_ECJPAKE_C**

Elliptic curve J-PAKE

*Found in: Component config > mbedTLS*

Enable ECJPAKE. Needed to use ECJPAKE-xxx TLS ciphersuites.

**Default value:**
- No (disabled)

**CONFIG_MBEDTLS_ECP_DP_SECP192R1_ENABLED**

Enable SECP192R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP192R1 Elliptic Curve.

**CONFIG_MBEDTLS_ECP_DP_SECP224R1_ENABLED**

Enable SECP224R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP224R1 Elliptic Curve.
CONFIG_MBEDTLS_ECP_DP_SECP256R1_ENABLED
Enable SECP256R1 curve

*Found in: Component config > mbedTLS*
Enable support for SECP256R1 Elliptic Curve.

**Default value:**
- Yes (enabled)

CONFIG_MBEDTLS_ECP_DP_SECP384R1_ENABLED
Enable SECP384R1 curve

*Found in: Component config > mbedTLS*
Enable support for SECP384R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP521R1_ENABLED
Enable SECP521R1 curve

*Found in: Component config > mbedTLS*
Enable support for SECP521R1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP192K1_ENABLED
Enable SECP192K1 curve

*Found in: Component config > mbedTLS*
Enable support for SECP192K1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP224K1_ENABLED
Enable SECP224K1 curve

*Found in: Component config > mbedTLS*
Enable support for SECP224K1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_SECP256K1_ENABLED
Enable SECP256K1 curve

*Found in: Component config > mbedTLS*
Enable support for SECP256K1 Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_BP256R1_ENABLED
Enable BP256R1 curve

*Found in: Component config > mbedTLS*
support for DP Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_BP384R1_ENABLED
Enable BP384R1 curve

*Found in: Component config > mbedTLS*
support for DP Elliptic Curve.
CONFIG_MBEDTLS_ECP_DP_BP512R1_ENABLED
Enable BP512R1 curve

Found in: Component config > mbedTLS

support for DP Elliptic Curve.

CONFIG_MBEDTLS_ECP_DP_CURVE25519_ENABLED
Enable CURVE25519 curve

Found in: Component config > mbedTLS

Enable support for CURVE25519 Elliptic Curve.

CONFIG_MBEDTLS_ECP_NIST_OPTIM
NIST ‘modulo p’ optimisations

Found in: Component config > mbedTLS

NIST ‘modulo p’ optimisations increase Elliptic Curve operation performance.

Disabling this option saves some code size.

Default value:
• Yes (enabled)

CONFIG_MBEDTLS_ECP_FIXED_POINT_OPTIM
Enable fixed-point multiplication optimisations

Found in: Component config > mbedTLS

This configuration option enables optimizations to speedup (about 3 ~ 4 times) the ECP fixed point multiplication using pre-computed tables in the flash memory. Disabling this configuration option saves flash footprint (about 29KB if all Elliptic Curve selected) in the application binary.

# end of Elliptic Curve options

Default value:
• Yes (enabled)

CONFIG_MBEDTLS_POLY1305_C
Poly1305 MAC algorithm

Found in: Component config > mbedTLS

Enable support for Poly1305 MAC algorithm.

Default value:
• No (disabled)

CONFIG_MBEDTLS_CHacha20_C
Chacha20 stream cipher

Found in: Component config > mbedTLS

Enable support for Chacha20 stream cipher.

Default value:
• No (disabled)
CONFIG_MBEDTLS_CHACHAPOLY_C
ChaCha20-Poly1305 AEAD algorithm

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_CHACHA20_C*
Enable support for ChaCha20-Poly1305 AEAD algorithm.

**Default value:**
- No (disabled) if `CONFIG_MBEDTLS_CHACHA20_C` && `CONFIG_MBEDTLS_POLY1305_C`

CONFIG_MBEDTLS_HKDF_C
HKDF algorithm (RFC 5869)

*Found in: Component config > mbedTLS*
Enable support for the Hashed Message Authentication Code (HMAC)-based key derivation function (HKDF).

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_THREADING_C
Enable the threading abstraction layer

*Found in: Component config > mbedTLS*
If you do intend to use contexts between threads, you will need to enable this layer to prevent race conditions.

**Default value:**
- No (disabled)

CONFIG_MBEDTLS_THREADING_ALT
Enable threading alternate implementation

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_THREADING_C*
Enable threading alt to allow your own alternate threading implementation.

**Default value:**
- Yes (enabled) if `CONFIG_MBEDTLS_THREADING_C`

CONFIG_MBEDTLS_THREADING_PTHREAD
Enable threading pthread implementation

*Found in: Component config > mbedTLS > CONFIG_MBEDTLS_THREADING_C*
Enable the pthread wrapper layer for the threading layer.

**Default value:**
- No (disabled) if `CONFIG_MBEDTLS_THREADING_C`

CONFIG_MBEDTLS_LARGE_KEYSOFTWARE_MPI
Fallback to software implementation for larger MPI values

*Found in: Component config > mbedTLS*
Fallback to software implementation for RSA key lengths larger than SOC_RSA_MAX_BIT_LEN. If this is not active then the ESP will be unable to process keys greater than SOC_RSA_MAX_BIT_LEN.
Default value:
• No (disabled)

**CONFIG_MBEDTLS_USE_CRYPTO_ROM_IMPL**

Use ROM implementation of the crypto algorithm

*Found in: Component config > mbedTLS*

Enable this flag to use mbedTLS crypto algorithm from ROM instead of ESP-IDF.

This configuration option saves flash footprint in the application binary. Note that the version of mbedTLS crypto algorithm library in ROM is v2.16.12. We have done the security analysis of the mbedTLS revision in ROM (v2.16.12) and ensured that affected symbols have been patched (removed). If in the future mbedTLS revisions there are security issues that also affects the version in ROM (v2.16.12) then we shall patch the relevant symbols. This would increase the flash footprint and hence care must be taken to keep some reserved space for the application binary in flash layout.

Default value:
• No (disabled) if ESP_ROM_HAS_MBEDTLS_CRYPTO_LIB && CONFIG_IDF_EXPERIMENTAL_FEATURES

**ESP-MQTT Configurations**  Contains:

• CONFIG_MQTT_CUSTOM_OUTBOX
• CONFIG_MQTT_TRANSPORT_SSL
• CONFIG_MQTT_TRANSPORT_WEBSOCKET
• CONFIG_MQTT_PROTOCOL_311
• CONFIG_MQTT_PROTOCOL_5
• CONFIG_MQTT_TASK_CORE_SELECTION_ENABLED
• CONFIG_MQTT_USE_CUSTOM_CONFIG
• CONFIG_MQTT_OUTBOX_EXPIRED_TIMEOUT_MS
• CONFIG_MQTT_REPORT_DELETED_MESSAGES
• CONFIG_MQTT_SKIP_PUBLISH_IF_DISCONNECTED
• CONFIG_MQTT_OUTBOX_DATA_ON_EXTERNAL_MEMORY
• CONFIG_MQTT_MSG_ID_INCREMENTAL

**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

Default value:
• Yes (enabled)

**CONFIG_MQTT_PROTOCOL_5**

Enable MQTT protocol 5.0

*Found in: Component config > ESP-MQTT Configurations*

If not, this library will not support MQTT 5.0

Default value:
• No (disabled)
CONFIG_MQTT_TRANSPORT_SSL
Enable MQTT over SSL

*Found in: Component config > ESP-MQTT Configurations*

Enable MQTT transport over SSL with mbedTLS

**Default value:**
- Yes (enabled)

CONFIG_MQTT_TRANSPORT_WEBSOCKET
Enable MQTT over Websocket

*Found in: Component config > ESP-MQTT Configurations*

Enable MQTT transport over Websocket.

**Default value:**
- Yes (enabled)

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.

**Default value:**
- Yes (enabled)

CONFIG_MQTT_MSG_ID_INCREMENTAL
Use Incremental Message Id

*Found in: Component config > ESP-MQTT Configurations*

Set this to true for the message id (2.3.1 Packet Identifier) to be generated as an incremental number rather than a random value (used by default).

**Default value:**
- No (disabled)

CONFIG_MQTT_SKIP_PUBLISH_IF_DISCONNECTED
Skip publish if disconnected

*Found in: Component config > ESP-MQTT Configurations*

Set this to true to avoid publishing (enqueueing messages) if the client is disconnected. The MQTT client tries to publish all messages by default, even in the disconnected state (where the qos1 and qos2 packets are stored in the internal outbox to be published later) The MQTT_SKIP_PUBLISH_IF_DISCONNECTED option allows applications to override this behaviour and not enqueue publish packets in the disconnected state.

**Default value:**
- No (disabled)
CONFIG_MQTT_REPORT_DELETED_MESSAGES
Report deleted messages

*Found in: Component config > ESP-MQTT Configurations*

Set this to true to post events for all messages which were deleted from the outbox before being correctly sent and confirmed.

**Default value:**
- No (disabled)

CONFIG_MQTT_USE_CUSTOM_CONFIG
MQTT Using custom configurations

*Found in: Component config > ESP-MQTT Configurations*

Custom MQTT configurations.

**Default value:**
- No (disabled)

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

**Default value:**
- 1883 if `CONFIG_MQTT_USE_CUSTOM_CONFIG`

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

**Default value:**
- 8883 if `CONFIG_MQTT_USE_CUSTOM_CONFIG` and `CONFIG_MQTT_TRANSPORT_SSL`

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

**Default value:**
- 80 if `CONFIG_MQTT_USE_CUSTOM_CONFIG` and `CONFIG_MQTT_TRANSPORT_WEBSOCKET`

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

**Default value:**
Chapter 2. API Reference

- 443 if \( \text{CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG} \) \&\& \( \text{CONFIG\_MQTT\_TRANSPORT\_WEBSOCKET} \) \&\& \( \text{CONFIG\_MQTT\_TRANSPORT\_WEBSOCKET\_SECURE} \)

**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

**Default value:**

- 1024 if \( \text{CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG} \)

**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

**Default value:**

- 6144 if \( \text{CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG} \)

**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

**Default value:**

- No (disabled) if \( \text{CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG} \)

**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.

**Default value:**

- 5 if \( \text{CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG} \)

**CONFIG\_MQTT\_POLL\_READ\_TIMEOUT\_MS**

MQTT transport poll read timeout

*Found in: Component config > ESP-MQTT Configurations > CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG*

Timeout when polling underlying transport for read.

**Default value:**

- 1000 if \( \text{CONFIG\_MQTT\_USE\_CUSTOM\_CONFIG} \)
**CONFIG_MQTT_EVENT_QUEUE_SIZE**

Number of queued events.

*Found in: Component config > ESP-MQTT Configurations > CONFIG_MQTT_USE_CUSTOM_CONFIG*

A value higher than 1 enables multiple queued events.

**Default value:**

- 1 if `CONFIG_MQTT_USE_CUSTOM_CONFIG`

**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 (CONFIG_MQTT_USE_CORE_0)
- Core 1 (CONFIG_MQTT_USE_CORE_1)

**CONFIG_MQTT_OUTBOX_DATA_ON_EXTERNAL_MEMORY**

Use external memory for outbox data

*Found in: Component config > ESP-MQTT Configurations*

Set to true to use external memory for outbox data.

**Default value:**

- No (disabled) if `CONFIG_MQTT_USE_CUSTOM_CONFIG`

**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). Note: Implementation of the custom outbox must be added to the mqtt component. These CMake commands could be used to append the custom implementation to lib-mqtt sources:

```
idf_component_get_property(mqtt mqtt COMPONENT_LIB) set_property(TARGET ${mqtt} PROPERTY SOURCES ${PROJECT_DIR}/custom_outbox.c APPEND)
```

**Default value:**

- No (disabled)

**CONFIG_MQTT_OUTBOX_EXPIRED_TIMEOUT_MS**

Outbox message expired timeout[ms]

*Found in: Component config > ESP-MQTT Configurations*

Messages which stays in the outbox longer than this value before being published will be discarded.
**Default value:**

- 30000 if `CONFIG_MQTT_USE_CUSTOM_CONFIG`

**Newlib** Contains:

- `CONFIG_NEWLIB_NANO_FORMAT`
- `CONFIG_NEWLIB_STDIN_LINE_ENDING`
- `CONFIG_NEWLIB_STDOUT_LINE_ENDING`
- `CONFIG_NEWLIB_TIME_SYSCALL`

**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 (CONFIG_NEWLIB_STDOUT_LINE_ENDING_CRLF)
- LF (CONFIG_NEWLIB_STDOUT_LINE_ENDING_LF)
- CR (CONFIG_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 (CONFIG_NEWLIB_STDIN_LINE_ENDING_CRLF)
- LF (CONFIG_NEWLIB_STDIN_LINE_ENDING_LF)
- CR (CONFIG_NEWLIB_STDIN_LINE_ENDING_CR)

**CONFIG_NEWLIB_NANO_FORMAT**

Enable ‘nano’ formatting options for printf/scanf family

*Found in: Component config > Newlib*
In most chips the 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 and the ROM contains functions from newlib-nano, the 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.

Some chips (e.g. ESP32-C6) has the full formatting versions of printf/scanf in ROM instead of the nano versions and in this building with newlib nano might actually increase the size of the binary. Which functions are present in ROM can be seen from ROM caps: ESP_ROM_HAS_NEWLIB_NANO_FORMAT and ESP_ROM_HAS_NEWLIB_NORMAL_FORMAT.

If you need 64-bit integer formatting support or C99 features, keep this option disabled.

**CONFIG_NEWLIB_TIME_SYSCALL**

Timers used for gettimeofday function

*Found in: Component config > Newlib*

This setting defines which hardware timers are used to implement ‘gettimeofday’ and ‘time’ functions in C library.

- **If both high-resolution (systimer for all targets except ESP32) 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 (systimer) 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 (CONFIG_NEWLIB_TIME_SYSCALL_USE_RTC_HRT)
- RTC (CONFIG_NEWLIB_TIME_SYSCALL_USE_RTC)
- High-resolution timer (CONFIG_NEWLIB_TIME_SYSCALL_USE_HRT)
- None (CONFIG_NEWLIB_TIME_SYSCALL_USE_NONE)

**NVS**

Contains:

- CONFIG_NVS_LEGACY_DUP_KEYS_COMPATIBILITY
- CONFIG_NVS_ENCRYPTION
- CONFIG_NVS_COMPATIBLE_PRE_V4_3_ENCRYPTION_FLAG
- CONFIG_NVS_ASSERT_ERROR_CHECK

**CONFIG_NVS_ENCRYPTION**

Enable NVS encryption

*Found in: Component config > NVS*
This option enables encryption for NVS. When enabled, XTS-AES is used to encrypt the complete NVS data, except the page headers. It requires XTS encryption keys to be stored in an encrypted partition (enabling flash encryption is mandatory here) or to be derived from an HMAC key burnt in eFuse.

**Default value:**
- Yes (enabled) if `CONFIG_SECURE_FLASH_ENC_ENABLED`

**CONFIG_NVS_COMPATIBLE_PRE_V4_3_ENCRYPTION_FLAG**

NVS partition encrypted flag compatible with ESP-IDF before v4.3

*Found in: Component config > NVS*

Enabling this will ignore “encrypted” flag for NVS partitions. NVS encryption scheme is different than hardware flash encryption and hence it is not recommended to have “encrypted” flag for NVS partitions. This was not being checked in pre v4.3 IDF. Hence, if you have any devices where this flag is kept enabled in partition table then enabling this config will allow to have same behavior as pre v4.3 IDF.

**CONFIG_NVS_ASSERT_ERROR_CHECK**

Use assertions for error checking.

*Found in: Component config > NVS*

This option switches error checking type between assertions (y) or return codes (n).

**Default value:**
- No (disabled)

**CONFIG_NVS_LEGACY_DUP_KEYS_COMPATIBILITY**

Enable legacy nvs_set function behavior when same key is reused with different data types

*Found in: Component config > NVS*

Enabling this option will switch the nvs_set() family of functions to the legacy mode: when called repeatedly with the same key but different data type, the existing value in the NVS remains active and the new value is just stored, actually not accessible through corresponding nvs_get() call for the key given. Use this option only when your application relies on such NVS API behaviour.

**Default value:**
- No (disabled)

**NVS Security Provider**

Contains:
- `CONFIG_NVS_SEC_HMAC_EFUSE_KEY_ID`
- `CONFIG_NVS_SEC_KEY_PROTECTION_SCHEME`

**CONFIG_NVS_SEC_KEY_PROTECTION_SCHEME**

NVS Encryption: Key Protection Scheme

*Found in: Component config > NVS Security Provider*

This choice defines the default NVS encryption keys protection scheme; which will be used for the default NVS partition. Users can use the corresponding scheme registration APIs to register other schemes for the default as well as other NVS partitions.

Available options:
• Using Flash Encryption (CONFIG_NVS_SEC_KEY_PROTECT_USING_FLASH_ENC)
  Protect the NVS Encryption Keys using Flash Encryption Requires a separate
  ‘nvs_keys’ partition (which will be encrypted by flash encryption) for storing the NVS
  encryption keys
• Using HMAC peripheral (CONFIG_NVS_SEC_KEY_PROTECT_USING_HMAC)
  Derive and protect the NVS Encryption Keys using the HMAC peripheral Re-
  quires the specified eFuse block (NVS_SEC_HMAC_EFUSE_KEY_ID or the
  v2 API argument) to be empty or pre-written with a key with the purpose
  ESP_EFUSE_KEY_PURPOSE_HMAC_UP

CONFIG_NVS_SEC_HMAC_EFUSE_KEY_ID

eFuse key ID storing the HMAC key

*Found in: Component config > NVS Security Provider*

eFuse block key ID storing the HMAC key for deriving the NVS encryption keys

Note: The eFuse block key ID required by the HMAC scheme (CON-
FIG_NVS_SEC_KEY_PROTECT_USING_HMAC) is set using this config when the default
NVS partition is initialized with nvs_flash_init(). The eFuse block key ID can also be set at runtime by
passing the appropriate value to the NVS security scheme registration APIs.

*Range:*
  • from 0 to 6 if CONFIG_NVS_SEC_KEY_PROTECT_USING_HMAC

*Default value:*
  • 6 if CONFIG_NVS_SEC_KEY_PROTECT_USING_HMAC

OpenThread  Contains:

- CONFIG_OPENTHREAD_PLATFORM_MSGPOOL_MANAGEMENT
- CONFIG_OPENTHREAD_DEVICE_TYPE
- CONFIG_OPENTHREAD_RADIO_TYPE
- CONFIG_OPENTHREAD_BORDER_ROUTER
- CONFIG_OPENTHREAD_COMMISSIONER
- CONFIG_OPENTHREAD_CSL_DEBUG_ENABLE
- CONFIG_OPENTHREAD_CSL_ENABLE
- CONFIG_OPENTHREAD_DIAG
- CONFIG_OPENTHREAD_DNS_CLIENT
- CONFIG_OPENTHREAD_DUA_ENABLE
- CONFIG_OPENTHREAD_JOINER
- CONFIG_OPENTHREAD_LINK_METRICS
- CONFIG_OPENTHREAD_MACFILTER_ENABLE
- CONFIG_OPENTHREAD_CLI
- CONFIG_OPENTHREAD_RADIO_STATS_ENABLE
- CONFIG_OPENTHREAD_SRP_CLIENT
- CONFIG_OPENTHREAD_TIME_SYNC
- CONFIG_OPENTHREAD_ENABLED
- CONFIG_OPENTHREAD_XTL_ACCURACY
- CONFIG_OPENTHREAD_CSL_UNCERTAIN
- CONFIG_OPENTHREAD_CSL_ACCURACY
- CONFIG_OPENTHREAD_NUM_MESSAGE_BUFFERS
- CONFIG_OPENTHREAD_RCP_TRANSPORT
- CONFIG_OPENTHREAD_MLE_MAX_CHILDREN
- CONFIG_OPENTHREAD_TMF_ADDR_CACHE_ENTRIES
- CONFIG_OPENTHREAD_SPINEL_RX_FRAME_BUFFER_SIZE
- CONFIG_OPENTHREAD_UART_BUFFER_SIZE
  *Thread Operational Dataset*
Chapter 2. API Reference

CONFIG_OPENTHREAD_ENABLED
OpenThread

*Found in: Component config > OpenThread*

Select this option to enable OpenThread and show the submenu with OpenThread configuration choices.

**Default value:**
- No (disabled)

CONFIG_OPENTHREAD_LOG_LEVEL_DYNAMIC
Enable dynamic log level control

*Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED*

Select this option to enable dynamic log level control for OpenThread

**Default value:**
- Yes (enabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_CONSOLE_TYPE
OpenThread console type

*Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED*

Select OpenThread console type

**Available options:**
- OpenThread console type UART (`CONFIG_OPENTHREAD_CONSOLE_TYPE_UART`)
- OpenThread console type USB Serial/JTAG Controller (`CONFIG_OPENTHREAD_CONSOLE_TYPE_USB_SERIAL_JTAG`)

CONFIG_OPENTHREAD_LOG_LEVEL
OpenThread log verbosity

*Found in: Component config > OpenThread > CONFIG_OPENTHREAD_ENABLED*

Select OpenThread log level.

**Available options:**
- No logs (`CONFIG_OPENTHREAD_LOG_LEVEL_NONE`)
- Error logs (`CONFIG_OPENTHREAD_LOG_LEVEL_CRIT`)
- Warning logs (`CONFIG_OPENTHREAD_LOG_LEVEL_WARN`)
- Notice logs (`CONFIG_OPENTHREAD_LOG_LEVEL_NOTE`)
- Info logs (`CONFIG_OPENTHREAD_LOG_LEVEL_INFO`)
- Debug logs (`CONFIG_OPENTHREAD_LOG_LEVEL_DEBUG`)

Thread Operational Dataset

Contains:
- `CONFIG_OPENTHREAD_NETWORK_EXTPANID`
- `CONFIG_OPENTHREAD_MESH_LOCAL_PREFIX`
- `CONFIG_OPENTHREAD_NETWORK_CHANNEL`
- `CONFIG_OPENTHREAD_NETWORK_MASTERKEY`
- `CONFIG_OPENTHREAD_NETWORK_NAME`
- `CONFIG_OPENTHREAD_NETWORK_PANID`
- `CONFIG_OPENTHREAD_NETWORK_PSKC`
CONFIG_OPENTHREAD_NETWORK_NAME
OpenThread network name

*Found in:* Component config > OpenThread > Thread Operational Dataset

*Default value:*
  - “OpenThread-ESP”

CONFIG_OPENTHREAD_MESH_LOCAL_PREFIX
OpenThread mesh local prefix, format `<address>/<plen>`

*Found in:* Component config > OpenThread > Thread Operational Dataset

A string in the format “`<address>/<plen>`”, where `<address>` is an IPv6 address and `<plen>` is a prefix length. For example “fd00:db8:a0:0::/64”

*Default value:*
  - “fd00:db8:a0:0::/64”

CONFIG_OPENTHREAD_NETWORK_CHANNEL
OpenThread network channel

*Found in:* Component config > OpenThread > Thread Operational Dataset

*Range:*
  - from 11 to 26

*Default value:*
  - 15

CONFIG_OPENTHREAD_NETWORK_PANID
OpenThread network pan id

*Found in:* Component config > OpenThread > Thread Operational Dataset

*Range:*
  - from 0 to 0xFFFFE

*Default value:*
  - “0x1234”

CONFIG_OPENTHREAD_NETWORK_EXTPANID
OpenThread extended pan id

*Found in:* Component config > OpenThread > Thread Operational Dataset

The OpenThread network extended pan id in hex string format

*Default value:*
  - dead00beef00cafe

CONFIG_OPENTHREAD_NETWORK_MASTERKEY
OpenThread network key

*Found in:* Component config > OpenThread > Thread Operational Dataset

The OpenThread network network key in hex string format

*Default value:*
  - 00112233445566778899aabbccddeeff
**CONFIG_OPENTHREAD_NETWORK_PSKC**

OpenThread pre-shared commissioner key

*Found in: Component config > OpenThread > Thread Operational Dataset*

The OpenThread pre-shared commissioner key in hex string format

**Default value:**
- 104810e2315100afd6be9215a6bfac53

**CONFIG_OPENTHREAD_RADIO_TYPE**

Config the Thread radio type

*Found in: Component config > OpenThread*

Configure how OpenThread connects to the 15.4 radio

Available options:

- Native 15.4 radio (CONFIG_OPENTHREAD_RADIO_NATIVE)
  Select this to use the native 15.4 radio.
- Connect via UART (CONFIG_OPENTHREAD_RADIO_SPINEL_UART)
  Select this to connect to a Radio Co-Processor via UART.
- Connect via SPI (CONFIG_OPENTHREAD_RADIO_SPINEL_SPI)
  Select this to connect to a Radio Co-Processor via SPI.

**CONFIG_OPENTHREAD_DEVICE_TYPE**

Config the Thread device type

*Found in: Component config > OpenThread*

OpenThread can be configured to different device types (FTD, MTD, Radio)

Available options:

- Full Thread Device (CONFIG_OPENTHREAD_FTD)
  Select this to enable Full Thread Device which can act as router and leader in a Thread network.
- Minimal Thread Device (CONFIG_OPENTHREAD_MTD)
  Select this to enable Minimal Thread Device which can only act as end device in a Thread network. This will reduce the code size of the OpenThread stack.
- Radio Only Device (CONFIG_OPENTHREAD_RADIO)
  Select this to enable Radio Only Device which can only forward 15.4 packets to the host. The OpenThread stack will be run on the host and OpenThread will have minimal footprint on the radio only device.

**CONFIG_OPENTHREAD_RCP_TRANSPORT**

The RCP transport type

*Found in: Component config > OpenThread*

Available options:

- UART RCP (CONFIG_OPENTHREAD_RCP_UART)
  Select this to enable UART connection to host.
- SPI RCP (CONFIG_OPENTHREAD_RCP_SPI)
  Select this to enable SPI connection to host.
CONFIG_OPENTHREAD_CLI

Enable Openthread Command-Line Interface

*Found in: Component config > OpenThread*

Select this option to enable Command-Line Interface in OpenThread.

**Default value:**
- Yes (enabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_DIAG

Enable diag

*Found in: Component config > OpenThread*

Select this option to enable Diag in OpenThread. This will enable diag mode and a series of diag commands in the OpenThread command line. These commands allow users to manipulate low-level features of the storage and 15.4 radio.

**Default value:**
- Yes (enabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_COMMISSIONER

Enable Commissioner

*Found in: Component config > OpenThread*

Select this option to enable commissioner in OpenThread. This will enable the device to act as a commissioner in the Thread network. A commissioner checks the pre-shared key from a joining device with the Thread commissioning protocol and shares the network parameter with the joining device upon success.

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_COMM_MAX_JOINER_ENTRIES

The size of max commissioning joiner entries

*Found in: Component config > OpenThread > CONFIG_OPENTHREAD_COMMISSIONER*

**Range:**
- from 2 to 50 if `CONFIG_OPENTHREAD_COMMISSIONER`

**Default value:**
- 2 if `CONFIG_OPENTHREAD_COMMISSIONER`

CONFIG_OPENTHREAD_JOINER

Enable Joiner

*Found in: Component config > OpenThread*

Select this option to enable Joiner in OpenThread. This allows a device to join the Thread network with a pre-shared key using the Thread commissioning protocol.

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_ENABLED`
CONFIG_OPENTHREAD_SRП_CLIENT
Enable SRP Client

*Found in: Component config > OpenThread*

Select this option to enable SRP Client in OpenThread. This allows a device to register SRP services to SRP Server.

**Default value:**
- Yes (enabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_SRП_CLIENT_MAX_SERVICES
Specifies number of service entries in the SRP client service pool

*Found in: Component config > OpenThread > CONFIG_OPENTHREAD_SRП_CLIENT*

Set the max buffer size of service entries in the SRP client service pool.

**Range:**
- from 2 to 20 if `CONFIG_OPENTHREAD_SRП_CLIENT`

**Default value:**
- 5 if `CONFIG_OPENTHREAD_SRП_CLIENT`

CONFIG_OPENTHREAD_DNS_CLIENT
Enable DNS Client

*Found in: Component config > OpenThread*

Select this option to enable DNS Client in OpenThread.

**Default value:**
- Yes (enabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_BORDER_ROUTER
Enable Border Router

*Found in: Component config > OpenThread*

Select this option to enable border router features in OpenThread.

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_PLATFORM_MSGPOOL_MANAGEMENT
Allocate message pool buffer from PSRAM

*Found in: Component config > OpenThread*

If enabled, the message pool is managed by platform defined logic.

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_ENABLED` \&\& (CONFIG_SPIRAM_USE_CAPS_ALLOC || CONFIG_SPIRAM_USE_MALLOC)

CONFIG_OPENTHREAD_NUM_MESSAGE_BUFFERS
The number of openthread message buffers

*Found in: Component config > OpenThread*

**Range:**
Chapter 2. API Reference

- from 10 to 8191 if `CONFIG_OPENTHREAD_PLATFORM_MSGPOOL_MANAGEMENT` && `CONFIG_OPENTHREAD_ENABLED`

  **Default value:**
  - 65 if `CONFIG_OPENTHREAD_ENABLED`

**CONFIG_OPENTHREAD_SPINEL_RX_FRAME_BUFFER_SIZE**

The size of openthread spinel rx frame buffer

  * Found in: Component config > OpenThread
  * Range:
    - from 512 to 8192 if `CONFIG_OPENTHREAD_ENABLED`
  * Default value:
    - 1024 if `CONFIG_OPENTHREAD_ENABLED`

**CONFIG_OPENTHREAD_MLE_MAX_CHILDREN**

The size of max MLE children entries

  * Found in: Component config > OpenThread
  * Range:
    - from 5 to 50 if `CONFIG_OPENTHREAD_ENABLED`
  * Default value:
    - 10 if `CONFIG_OPENTHREAD_ENABLED`

**CONFIG_OPENTHREAD_TMF_ADDR_CACHE_ENTRIES**

The size of max TMF address cache entries

  * Found in: Component config > OpenThread
  * Range:
    - from 5 to 50 if `CONFIG_OPENTHREAD_ENABLED`
  * Default value:
    - 20 if `CONFIG_OPENTHREAD_ENABLED`

**CONFIG_OPENTHREAD_DNS64_CLIENT**

Use dns64 client

  * Found in: Component config > OpenThread
  * Select this option to acquire NAT64 address from dns servers.

  * Default value:
    - No (disabled) if `CONFIG_OPENTHREAD_ENABLED` && `CONFIG_LWIP_IPV4`

**CONFIG_OPENTHREAD_DNS_SERVER_ADDR**

DNS server address (IPv4)

  * Found in: Component config > OpenThread > CONFIG_OPENTHREAD_DNS64_CLIENT
  * Set the DNS server IPv4 address.

  * Default value:
    - “8.8.8.8” if `CONFIG_OPENTHREAD_DNS64_CLIENT`
CONFIG_OPENTHREAD_UART_BUFFER_SIZE
The uart received buffer size of openthread

Found in: Component config > OpenThread
Set the OpenThread UART buffer size.

Range:
  • from 128 to 1024 if CONFIG_OPENTHREAD_ENABLED

Default value:
  • 256 if CONFIG_OPENTHREAD_ENABLED

CONFIG_OPENTHREAD_LINK_METRICS
Enable link metrics feature

Found in: Component config > OpenThread
Select this option to enable link metrics feature

Default value:
  • No (disabled) if CONFIG_OPENTHREAD_ENABLED

CONFIG_OPENTHREAD_MACFILTER_ENABLE
Enable mac filter feature

Found in: Component config > OpenThread
Select this option to enable mac filter feature

Default value:
  • No (disabled) if CONFIG_OPENTHREAD_ENABLED

CONFIG_OPENTHREAD_CSL_ENABLE
Enable CSL feature

Found in: Component config > OpenThread
Select this option to enable CSL feature

Default value:
  • No (disabled) if CONFIG_OPENTHREAD_ENABLED

CONFIG_OPENTHREAD_XTAL_ACCURACY
The accuracy of the XTAL

Found in: Component config > OpenThread
The device’s XTAL accuracy, in ppm.

Default value:
  • 130

CONFIG_OPENTHREAD_CSL_ACCURACY
The current CSL rx/tx scheduling drift, in units of ± ppm

Found in: Component config > OpenThread
The current accuracy of the clock used for scheduling CSL operations

Default value:
  • 1 if CONFIG_OPENTHREAD_CSL_ENABLE
Chapter 2. API Reference

CONFIG_OPENTHREAD_CSL_UNCERTAIN

The CSL Uncertainty in units of 10 us.

*Found in: Component config > OpenThread*

The fixed uncertainty of the Device for scheduling CSL Transmissions in units of 10 microseconds.

**Default value:**
- 1 if `CONFIG_OPENTHREAD_CSL_ENABLE`

CONFIG_OPENTHREAD_CSL_DEBUG_ENABLE

Enable CSL debug

*Found in: Component config > OpenThread*

Select this option to set rx on when sleep in CSL feature, only for debug

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_CSL_ENABLE`

CONFIG_OPENTHREAD_DUA_ENABLE

Enable Domain Unicast Address feature

*Found in: Component config > OpenThread*

Only used for Thread1.2 certification

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_TIME_SYNC

Enable the time synchronization service feature

*Found in: Component config > OpenThread*

Select this option to enable time synchronization feature, the devices in the same Thread network could sync to the same network time.

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_ENABLED`

CONFIG_OPENTHREAD_RADIO_STATS_ENABLE

Enable Radio Statistics feature

*Found in: Component config > OpenThread*

Select this option to enable the radio statistics feature, you can use radio command to print some radio Statistics informations.

**Default value:**
- No (disabled) if `CONFIG_OPENTHREAD_FTD || CONFIG_OPENTHREAD_MTD`

**Protocomm** Contains:

- `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0`
- `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1`
- `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2`
CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0

Support protocomm security version 0 (no security)

*Found in: Component config > Protocomm*

Enable support of security version 0. Disabling this option saves some code size. Consult the Enabling protocomm security version section of the Protocomm documentation in ESP-IDF Programming guide for more details.

**Default value:**
- Yes (enabled)

CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1

Support protocomm security version 1 (Curve25519 key exchange + AES-CTR encryption/decryption)

*Found in: Component config > Protocomm*

Enable support of security version 1. Disabling this option saves some code size. Consult the Enabling protocomm security version section of the Protocomm documentation in ESP-IDF Programming guide for more details.

**Default value:**
- Yes (enabled)

CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2

Support protocomm security version 2 (SRP6a-based key exchange + AES-GCM encryption/decryption)

*Found in: Component config > Protocomm*

Enable support of security version 2. Disabling this option saves some code size. Consult the Enabling protocomm security version section of the Protocomm documentation in ESP-IDF Programming guide for more details.

**Default value:**
- Yes (enabled)

**PThreads**  Contains:

- `CONFIG_PTHREAD_TASK_NAME_DEFAULT`
- `CONFIG_PTHREAD_TASK_CORE_DEFAULT`
- `CONFIG_PTHREAD_TASK_PRIO_DEFAULT`
- `CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT`
- `CONFIG_PTHREAD_STACK_MIN`

CONFIG_PTHREAD_TASK_PRIO_DEFAULT

Default task priority

*Found in: Component config > PThreads*

Priority used to create new tasks with default pthread parameters.

**Range:**
- from 0 to 255

**Default value:**
- 5
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.

**Default value:**
- 3072

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

**Default value:**
- 768

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 (CONFIG_PTHREAD_DEFAULT_CORE_NO_AFFINITY)
- Core 0 (CONFIG_PTHREAD_DEFAULT_CORE_0)
- Core 1 (CONFIG_PTHREAD_DEFAULT_CORE_1)

CONFIG_PTHREAD_TASK_NAME_DEFAULT

Default name of pthreads

*Found in: Component config > PThreads*

The default name of pthreads.

**Default value:**
- “pthread”

SoC Settings  Contains:
- MMU Config

MMU Config

Main Flash configuration  Contains:
- Optional and Experimental Features (READ DOCS FIRST)
- SPI Flash behavior when brownout

SPI Flash behavior when brownout  Contains:
- CONFIG_SPI_FLASH_BROWNOUT_RESET_XMC
CONFIG_SPI_FLASH_BROWNOUT_RESET_XMC

Enable sending reset when brownout for XMC flash chips

*Found in: Component config > Main Flash configuration > SPI Flash behavior when brownout*

When this option is selected, the patch will be enabled for XMC. Follow the recommended flow by XMC for better stability.

**DO NOT DISABLE UNLESS YOU KNOW WHAT YOU ARE DOING.**

Optional and Experimental Features (READ DOCS FIRST)  Contains:

- CONFIG_SPI_FLASH_AUTO_SUSPEND
- CONFIG_SPI_FLASH_SUSPEND_TSUS_VAL_US
- CONFIG_SPI_FLASH_HPM_DC

CONFIG_SPI_FLASH_HPM_DC

Support HPM using DC (READ DOCS FIRST)

*Found in: Component config > Main Flash configuration > Optional and Experimental Features (READ DOCS FIRST)*

This feature needs your bootloader to be compiled DC-aware (BOOTLOADER_FLASH_DC_AWARE=y). Otherwise the chip will not be able to boot after a reset.

Available options:

- Auto (Enable when bootloader support enabled (BOOTLOADER_FLASH_DC_AWARE=y)) (CONFIG_SPI_FLASH_HPM_DC_AUTO)
- Disable (READ DOCS FIRST) (CONFIG_SPI_FLASH_HPM_DC_DISABLE)

CONFIG_SPI_FLASH_AUTO_SUSPEND

Auto suspend long erase/write operations (READ DOCS FIRST)

*Found in: Component config > Main Flash configuration > Optional and Experimental Features (READ DOCS FIRST)*

This option is disabled by default because it is supported only for specific flash chips and for specific Espressif chips. To evaluate if you can use this feature refer to Optional Features for Flash > Auto Suspend & Resume of the ESP-IDF Programming Guide.

**CAUTION:** If you want to OTA to an app with this feature turned on, please make sure the bootloader has the support for it. (later than IDF v4.3)

If you are using an official Espressif module, please contact Espressif Business support to check if the module has the flash that support this feature installed. Also refer to Concurrency Constraints for Flash on SPI1 > Flash Auto Suspend Feature before enabling this option.

CONFIG_SPI_FLASH_SUSPEND_TSUS_VAL_US

SPI flash tSUS value (refer to chapter AC CHARACTERISTICS)

*Found in: Component config > Main Flash configuration > Optional and Experimental Features (READ DOCS FIRST)*

This config is used for setting Tsus parameter. Tsus means CS# high to next command after suspend. You can refer to the chapter of AC CHARACTERISTICS of flash datasheet.
SPI Flash driver  Contains:

- Auto-detect flash chips
- CONFIG_SPI_FLASH_BYPASS_BLOCK_ERASE
- CONFIG_SPI_FLASH_ENABLE_ENCRYPTED_READ_WRITE
- CONFIG_SPI_FLASH_ENABLE_COUNTERS
- CONFIG_SPI_FLASH_ROM_DRIVER_PATCH
- CONFIG_SPI_FLASH_YIELD_DURING_ERASE
- CONFIG_SPI_FLASH_CHECK_ERASE_TIMEOUT_DISABLED
- CONFIG_SPI_FLASH_WRITE_CHUNK_SIZE
- CONFIG_SPI_FLASH_OVERRIDE_CHIP_DRIVER_LIST
- CONFIG_SPI_FLASH_SIZE_OVERRIDE
- CONFIG_SPI_FLASH_ROM_IMPL
- CONFIG_SPI_FLASH_VERIFY_WRITE
- CONFIG_SPI_FLASH_DANGEROUS_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:

- esp_flash_reset_counters
- esp_flash_dump_counters
- esp_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 should be enabled, if any one of the following is true: (1) need to write to flash on ESP32-D2WD; (2) main SPI flash is connected to non-default pins; (3) main SPI flash chip is manufactured by ISSI.

### CONFIG_SPI_FLASH_ROM_IMPL
Use esp_flash implementation in ROM

*Found in:* Component config > SPI Flash driver

Enable this flag to use new SPI flash driver functions from ROM instead of ESP-IDF.

If keeping this as “n” in your project, you will have less free IRAM. But you can use all of our flash features.

If making this as “y” in your project, you will increase free IRAM. But you may miss out on some flash features and support for new flash chips.

Currently the ROM cannot support the following features:

- SPI_FLASH_AUTO_SUSPEND (C3, S3)

### 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 (CONFIG_SPI_FLASH_DANGEROUS_WRITE_ABORTS)
- Fails (CONFIG_SPI_FLASH_DANGEROUS_WRITE_FAILS)
- Allowed (CONFIG_SPI_FLASH_DANGEROUS_WRITE_ALLOWED)

### 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. Please use this configuration together with SPI\_FLASH\_ERASE\_YIELD\_DURATION\_MS and SPI\_FLASH\_ERASE\_YIELD\_TICKS after carefully checking flash datasheet to avoid a watchdog timeout. For more information, please check SPI Flash API reference documentation under section OS Function.

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) for an erase operation

 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_WRITE_CHUNK_SIZE

Flash write chunk size

 FOUND in: Component config > SPI Flash driver

Flash write is broken down in terms of multiple (smaller) write operations. This configuration options helps to set individual write chunk size, smaller value here ensures that cache (and non-IRAM resident interrupts) remains disabled for shorter duration.

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.

CONFIG_SPI_FLASH_CHECK_ERASE_TIMEOUT_DISABLED

Flash timeout checkout disabled

 FOUND in: Component config > SPI Flash driver

This option is helpful if you are using a flash chip whose timeout is quite large or unpredictable.

CONFIG_SPI_FLASH_OVERRIDE_CHIP_DRIVER_LIST

Override default chip driver list

 FOUND in: Component config > SPI Flash driver

This option allows the chip driver list to be customized, instead of using the default list provided by ESP-IDF.
When this option is enabled, the default list is no longer compiled or linked. Instead, the `default_registered_chips` structure must be provided by the user.

See example: custom_chip_driver under examples/storage for more details.

**Auto-detect flash chips**  
Contains:

- `CONFIG_SPI_FLASH_SUPPORT_BOYA_CHIP`
- `CONFIG_SPI_FLASH_SUPPORT_GD_CHIP`
- `CONFIG_SPI_FLASH_SUPPORT_ISSI_CHIP`
- `CONFIG_SPI_FLASH_SUPPORT_MXIC_CHIP`
- `CONFIG_SPI_FLASH_SUPPORT_TH_CHIP`
- `CONFIG_SPI_FLASH_SUPPORT_WINBOND_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_MXIC_CHIP**  
MXIC  
*Found in: Component config > SPI Flash driver > Auto-detect flash chips*  
Enable this to support auto detection of MXIC 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_SUPPORT_WINBOND_CHIP**  
Winbond  
*Found in: Component config > SPI Flash driver > Auto-detect flash chips*  
Enable this to support auto detection of Winbond 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_BOYA_CHIP**  
BOYA  
*Found in: Component config > SPI Flash driver > Auto-detect flash chips*  
Enable this to support auto detection of BOYA 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_TH_CHIP

TH

*Found in: Component config > SPI Flash driver > Auto-detect flash chips*

Enable this to support auto detection of TH 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_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:

- Debug Configuration
- CONFIG_SPIFFS_USE_MAGIC
- CONFIG_SPIFFS_GC_STATS
- CONFIG_SPIFFS_PAGE_CHECK
- CONFIG_SPIFFS_FOLLOW_SYMLINKS
- CONFIG_SPIFFS_MAX_PARTITIONS
- CONFIG_SPIFFS_USE_MTIME
- CONFIG_SPIFFS_GC_MAX_RUNS
- CONFIG_SPIFFS_OBJ_NAME_LEN
- CONFIG_SPIFFS_META_LENGTH
- SPIFFS Cache Configuration
- CONFIG_SPIFFS_PAGE_SIZE
- CONFIG_SPIFFS_MTIME_WIDE_64_BITS

CONFIG_SPIFFS_MAX_PARTITIONS

Maximum Number of Partitions

*Found in: Component config > SPIFFS Configuration*

Define maximum number of partitions that can be mounted.

**Range:**

- from 1 to 10

**Default value:**

- 3

SPIFFS Cache Configuration  Contains:

- CONFIG_SPIFFS_CACHE

CONFIG_SPIFFS_CACHE

Enable SPIFFS Cache

*Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration*

Enables/disables memory read caching of nucleus file system operations.

**Default value:**

- Yes (enabled)
Chapter 2. API Reference

**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.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- Yes (enabled)

**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.

**Range:**
- from 1 to 10000

**Default value:**
- 10

**CONFIG_SPIFFS_GC_STATS**

Enable SPIFFS GC Statistics

*Found in: Component config > SPIFFS Configuration*

Enable/disable statistics on gc. Debug/test purpose only.

**Default value:**
- No (disabled)
**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.

**Range:**
- from 256 to 1024

**Default value:**
- 256

**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.

**Range:**
- from 1 to 256

**Default value:**
- 32

**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- Yes (enabled)

**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.

**Default value:**
• Yes (enabled)

**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.

**Default value:**

• 4

**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.

**Default value:**

• Yes (enabled)

**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 64-bit time_t support.

**Default value:**

• No (disabled) if **CONFIG_SPIFFS_META_LENGTH** >= 8

**Debug Configuration**  Contains:

• **CONFIG_SPIFFS_DBG**
• **CONFIG_SPIFFS_API_DBG**
• **CONFIG_SPIFFS_CACHE_DBG**
• **CONFIG_SPIFFS_CHECK_DBG**
• **CONFIG_SPIFFS_TEST_VISUALISATION**
• **CONFIG_SPIFFS_GC_DBG**

**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.

**Default value:**

• No (disabled)
**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled)

**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.

**Default value:**
- No (disabled)

**TCP Transport**

Contains:

- **Websocket**

**Websocket**

Contains:

- **CONFIG_WS_TRANSPORT**
**CONFIG_WS_TRANSPORT**

Enable Websocket Transport

*Found in: Component config > TCP Transport > Websocket*

Enable support for creating websocket transport.

**Default value:**
- Yes (enabled)

**CONFIG_WS_BUFFER_SIZE**

Websocket transport buffer size

*Found in: Component config > TCP Transport > Websocket > CONFIG_WS_TRANSPORT*

Size of the buffer used for constructing the HTTP Upgrade request during connect

**Default value:**
- 1024

**CONFIG_WS_DYNAMIC_BUFFER**

Using dynamic websocket transport buffer

*Found in: Component config > TCP Transport > Websocket > CONFIG_WS_TRANSPORT*

If enable this option, websocket transport buffer will be freed after connection succeed to save more heap.

**Default value:**
- No (disabled)

**Ultra Low Power (ULP) Co-processor**

Contains:

- **CONFIG_ULP_COPROC_ENABLED**
- **ULP RISC-V Settings**

**CONFIG_ULP_COPROC_ENABLED**

Enable Ultra Low Power (ULP) Co-processor

*Found in: Component config > Ultra Low Power (ULP) Co-processor*

Enable this feature if you plan to use the ULP Co-processor. Once this option is enabled, further ULP co-processor configuration will appear in the menu.

**Default value:**
- No (disabled)

**CONFIG_ULP_COPROC_TYPE**

ULP Co-processor type

*Found in: Component config > Ultra Low Power (ULP) Co-processor > CONFIG_ULP_COPROC_ENABLED*

Choose the ULP Coprocessor type: ULP FSM (Finite State Machine) or ULP RISC-V.

Available options:

- ULP FSM (Finite State Machine) (CONFIG_ULP_COPROC_TYPE_FSM)
- ULP RISC-V (CONFIG_ULP_COPROC_TYPE_RISCV)
• LP core RISC-V (CONFIG_ULP_COPROC_TYPE_LP_CORE)

**CONFIG_ULP_COPROC_RESERVE_MEM**

RTC slow memory reserved for coprocessor

*Found in: Component config > Ultra Low Power (ULP) Co-processor > CONFIG_ULP_COPROC_ENABLED*

Bytes of memory to reserve for ULP Co-processor firmware & data. Data is reserved at the beginning of RTC slow memory.

**Range:**
- from 32 to 8176 if `CONFIG_ULP_COPROC_ENABLED`

**Default value:**
- 4096 if `CONFIG_ULP_COPROC_ENABLED`

**ULP RISC-V Settings**

Contains:
- `CONFIG_ULP_RISCV_UART_BAUDRATE`
- `CONFIG_ULP_RISCV_I2C_RW_TIMEOUT`

**CONFIG_ULP_RISCV_UART_BAUDRATE**

Baudrate used by the bitbanged ULP RISC-V UART driver

*Found in: Component config > Ultra Low Power (ULP) Co-processor > ULP RISC-V Settings*

The accuracy of the bitbanged UART driver is limited, it is not recommend to increase the value above 19200.

**Default value:**
- 9600 if `CONFIG_ULP_COPROC_TYPE_RISCV`

**CONFIG_ULP_RISCV_I2C_RW_TIMEOUT**

Set timeout for ULP RISC-V I2C transaction timeout in ticks.

*Found in: Component config > Ultra Low Power (ULP) Co-processor > ULP RISC-V Settings*

Set the ULP RISC-V I2C read/write timeout. Set this value to -1 if the ULP RISC-V I2C read and write APIs should wait forever. Please note that the tick rate of the ULP co-processor would be different than the OS tick rate of the main core and therefore can have different timeout value depending on which core the API is invoked on.

**Range:**
- from -1 to 4294967295 if `CONFIG_ULP_COPROC_TYPE_RISCV`

**Default value:**
- 500 if `CONFIG_ULP_COPROC_TYPE_RISCV`

**Unity unit testing library**

Contains:
- `CONFIGUNITY_ENABLE_COLOR`
- `CONFIGUNITY_ENABLE_IDF_TEST_RUNNER`
- `CONFIGUNITY_ENABLE_FIXTURE`
- `CONFIGUNITY_ENABLE_BACKTRACE_ON_FAIL`
- `CONFIGUNITY_ENABLE_64BIT`
- `CONFIGUNITY_ENABLE_DOUBLE`
- `CONFIGUNITY_ENABLE_FLOAT`
Chapter 2. API Reference

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.

**Default value:**
- Yes (enabled)

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.

**Default value:**
- Yes (enabled)

CONFIG_UNITY_ENABLE_64BIT

Support for 64-bit integer types

*Found in: Component config > Unity unit testing library*

If not set, assertions on 64-bit integer types will always fail. If this feature is enabled, take care not to pass pointers (which are 32 bit) to UNITY_ASSERT_EQUAL, as that will cause pointer-to-int-cast warnings.

**Default value:**
- No (disabled)

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.

**Default value:**
- No (disabled)

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.

**Default value:**
- Yes (enabled)
Chapter 2. API Reference

**CONFIGUNITY_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.

**Default value:**

- No (disabled)

**CONFIGUNITY_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.

**Default value:**

- No (disabled)

**USB-OTG**  Contains:

- **CONFIG_USB_HOST_HW_BUFFER_BIAS**
- **CONFIG_USB_HOST_CONTROL_TRANSFER_MAX_SIZE**
- **Root Hub configuration**

**CONFIG_USB_HOST_CONTROL_TRANSFER_MAX_SIZE**

Largest size (in bytes) of transfers to/from default endpoints

*Found in: Component config > USB-OTG*

Each USB device attached is allocated a dedicated buffer for its OUT/IN transfers to/from the device’s control endpoint. The maximum size of that buffer is determined by this option. The limited size of the transfer buffer have the following implications: - The maximum length of control transfers is limited - Device’s with configuration descriptors larger than this limit cannot be supported

**Default value:**

- 256

**CONFIG_USB_HOST_HW_BUFFER_BIAS**

Hardware FIFO size biasing

*Found in: Component config > USB-OTG*

The underlying hardware has size adjustable FIFOs to cache USB packets on reception (IN) or for transmission (OUT). The size of these FIFOs will affect the largest MPS (maximum packet size) and the maximum number of packets that can be cached at any one time. The hardware contains the following FIFOs: RX (for all IN packets), Non-periodic TX (for Bulk and Control OUT packets), and Periodic TX (for Interrupt and Isochronous OUT packets). This configuration option allows biasing the FIFO sizes towards a particular use case, which may be necessary for devices that have endpoints with large MPS. The MPS limits for each biasing are listed below:

Balanced: - IN (all transfer types), 408 bytes - OUT non-periodic (Bulk/Control), 192 bytes (i.e., 3 x 64 byte packets) - OUT periodic (Interrupt/Isochronous), 192 bytes

Bias IN: - IN (all transfer types), 600 bytes - OUT non-periodic (Bulk/Control), 64 bytes (i.e., 1 x 64 byte packets) - OUT periodic (Interrupt/Isochronous), 128 bytes
Bias Periodic OUT: - IN (all transfer types), 128 bytes - OUT non-periodic (Bulk/Control), 64 bytes (i.e., 1 x 64 byte packets) - OUT periodic (Interrupt/Isochronous), 600 bytes

Available options:

- Balanced (CONFIG_USB_HOST_HW_BUFFER_BIAS_BALANCED)
- Bias IN (CONFIG_USB_HOST_HW_BUFFER_BIAS_IN)
- Periodic OUT (CONFIG_USB_HOST_HW_BUFFER_BIAS_PERIODIC_OUT)

**Root Hub configuration** Contains:

- `CONFIG_USB_HOST_DEBOUNCE_DELAY_MS`
- `CONFIG_USB_HOST_RESET_HOLD_MS`
- `CONFIG_USB_HOST_RESET_RECOVERY_MS`
- `CONFIG_USB_HOST_SET_ADDR_RECOVERY_MS`

**CONFIG_USB_HOST_DEBOUNCE_DELAY_MS**

Debounce delay in ms

*Found in: Component config > USB-OTG > Root Hub configuration*

On connection of a USB device, the USB 2.0 specification requires a “debounce interval with a minimum duration of 100ms” to allow the connection to stabilize (see USB 2.0 chapter 7.1.7.3 for more details). During the debounce interval, no new connection/disconnection events are registered.

The default value is set to 250 ms to be safe.

**Default value:**
- 250

**CONFIG_USB_HOST_RESET_HOLD_MS**

Reset hold in ms

*Found in: Component config > USB-OTG > Root Hub configuration*

The reset signaling can be generated on any Hub or Host Controller port by request from the USB System Software. The USB 2.0 specification requires that “the reset signaling must be driven for a minimum of 10ms” (see USB 2.0 chapter 7.1.7.5 for more details). After the reset, the hub port will transition to the Enabled state (refer to Section 11.5).

The default value is set to 30 ms to be safe.

**Default value:**
- 30

**CONFIG_USB_HOST_RESET_RECOVERY_MS**

Reset recovery delay in ms

*Found in: Component config > USB-OTG > Root Hub configuration*

After a port stops driving the reset signal, the USB 2.0 specification requires that the “USB System Software guarantees a minimum of 10 ms for reset recovery” before the attached device is expected to respond to data transfers (see USB 2.0 chapter 7.1.7.3 for more details). The device may ignore any data transfers during the recovery interval.

The default value is set to 30 ms to be safe.

**Default value:**
- 30
**CONFIG_USB_HOST_SET_ADDR_RECOVERY_MS**

SetAddress() recovery time in ms

*Found in: Component config > USB-OTG > Root Hub configuration*

“After successful completion of the Status stage, the device is allowed a SetAddress() recovery interval of 2 ms. At the end of this interval, the device must be able to accept Setup packets addressed to the new address. Also, at the end of the recovery interval, the device must not respond to tokens sent to the old address (unless, of course, the old and new address is the same).” See USB 2.0 chapter 9.2.6.3 for more details.

The default value is set to 10 ms to be safe.

**Default value:**
- 10

**Virtual file system**  
Contains:

- **CONFIG_VFS_SUPPORT_IO**

**CONFIG_VFS_SUPPORT_IO**

Provide basic I/O functions

*Found in: Component config > Virtual file system*

If enabled, the following functions are provided by the VFS component.

open, close, read, write, pread, pwrite, lseek, fstat, fsync, ioctl, fcntl

Filesystem drivers can then be registered to handle these functions for specific paths.

Disabling this option can save memory when the support for these functions is not required.

Note that the following functions can still be used with socket file descriptors when this option is disabled:

close, read, write, ioctl, fcntl.

**Default value:**
- Yes (enabled)

**CONFIG_VFS_SUPPORT_DIR**

Provide directory related functions

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO*

If enabled, the following functions are provided by the VFS component.

stat, link, unlink, rename, utime, access, truncate, rmdir, mkdir, opendir, closedir, readdir, readdir_r, seekdir, telldir, rewinddir

Filesystem drivers can then be registered to handle these functions for specific paths.

Disabling this option can save memory when the support for these functions is not required.

**Default value:**
- Yes (enabled)

**CONFIG_VFS_SUPPORT_SELECT**

Provide select function

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO*

If enabled, select function is provided by the VFS component, and can be used on peripheral file descriptors (such as UART) and sockets at the same time.
Chapter 2. API Reference

If disabled, the default select implementation will be provided by LWIP for sockets only. Disabling this option can reduce code size if support for “select” on UART file descriptors is not required.

**CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT**

Suppress select() related debug outputs

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO > CONFIG_VFS_SUPPORT_SELECT*  

Select() related functions might produce an unconveniently 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.

**Default value:**  
- Yes (enabled)

**CONFIG_VFS_SELECT_IN_RAM**

Make VFS driver select() callbacks IRAM-safe

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO > CONFIG_VFS_SUPPORT_SELECT*  

If enabled, VFS driver select() callback function will be placed in IRAM.

**Default value:**  
- No (disabled)

**CONFIG_VFS_SUPPORT_TERMIOS**

Provide termios.h functions

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO*  

Disabling this option can save memory when the support for termios.h is not required.

**Default value:**  
- Yes (enabled)

**CONFIG_VFS_MAX_COUNT**

Maximum Number of Virtual Filesystems

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO*  

Define maximum number of virtual filesystems that can be registered.

**Range:**  
- from 1 to 20

**Default value:**  
- 8

**Host File System I/O (Semihosting)** Contains:

- `CONFIG_VFS_SEMIHOSTFS_MAX_MOUNT_POINTS`
**CONFIG_VFS_SEMIHOSTFS_MAX_MOUNT_POINTS**

Host FS: Maximum number of the host filesystem mount points

*Found in: Component config > Virtual file system > CONFIG_VFS_SUPPORT_IO > Host File System I/O (Semihosting)*

Define maximum number of host filesystem mount points.

**Default value:**

- 1

**Wear Levelling** Contains:

- CONFIG_WL_SECTOR_MODE
- CONFIG_WL_SECTOR_SIZE

**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 (CONFIG_WL_SECTOR_SIZE_512)
- 4096 (CONFIG_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 (CONFIG_WL_SECTOR_MODE_PERF)
- Safety (CONFIG_WL_SECTOR_MODE_SAFE)
**Wi-Fi Provisioning Manager** Contains:

- `CONFIG_WIFI_PROV_BLE_BONDING`
- `CONFIG_WIFI_PROV_BLE_SEC_CONN`
- `CONFIG_WIFI_PROV_BLE_FORCE_ENCRYPTION`
- `CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV`
- `CONFIG_WIFI_PROV_SCAN_MAX_ENTRIES`
- `CONFIG_WIFI_PROV_AUTOSTOP_TIMEOUT`
- `CONFIG_WIFI_PROV_STA_SCAN_METHOD`

**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.

**Range:**

- from 1 to 255

**Default value:**

- 16

**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.

**Range:**

- from 5 to 600

**Default value:**

- 30

**CONFIG_WIFI_PROV_BLE_BONDING**

Enable BLE bonding

*Found in: Component config > Wi-Fi Provisioning Manager*

This option is applicable only when provisioning transport is BLE.

**CONFIG_WIFI_PROV_BLE_SEC_CONN**

Enable BLE Secure connection flag

*Found in: Component config > Wi-Fi Provisioning Manager*

Used to enable Secure connection support when provisioning transport is BLE.

**Default value:**

- Yes (enabled) if `CONFIG_BT_NIMBLE_ENABLED`

**CONFIG_WIFI_PROV_BLE_FORCE_ENCRYPTION**

Force Link Encryption during characteristic Read / Write

*Found in: Component config > Wi-Fi Provisioning Manager*

Used to enforce link encryption when attempting to read / write characteristic
CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV

Keep BT on after provisioning is done

*Found in: Component config > Wi-Fi Provisioning Manager*

CONFIG_WIFI_PROV_DISCONNECT_AFTER_PROV

Terminate connection after provisioning is done

*Found in: Component config > Wi-Fi Provisioning Manager > CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV*

**Default value:**
- Yes (enabled) if `CONFIG_WIFI_PROV_KEEP_BLE_ON_AFTER_PROV`

CONFIG_WIFI_PROV_STA_SCAN_METHOD

Wifi Provisioning Scan Method

*Found in: Component config > Wi-Fi Provisioning Manager*

Available options:

- All Channel Scan (`CONFIG_WIFI_PROV_STA_ALL_CHANNEL_SCAN`)
  - Scan will end after scanning the entire channel. This option is useful in Mesh WiFi Systems.
- Fast Scan (`CONFIG_WIFI_PROV_STA_FAST_SCAN`)
  - Scan will end after an AP matching with the SSID has been detected.

CONFIG_IDF_EXPERIMENTAL_FEATURES

Make experimental features visible

*Found in:*

By enabling this option, ESP-IDF experimental feature options will be visible.

Note you should still enable a certain experimental feature option to use it, and you should read the corresponding risk warning and known issue list carefully.

Current experimental feature list:

- `CONFIG_ESPTOOLPY_FLASHFREQ_120M` & `CONFIG_ESPTOOLPY_FLASH_SAMPLE_MODE_DTR`
- `CONFIG_SPIRAM_SPEED_120M` & `CONFIG_SPIRAM_MODE_OCT`
- `CONFIG_BOOTLOADER_CACHE_32BIT_ADDR_QUAD_FLASH`
- `CONFIG_MBEDTLS_USE_CRYPTO_ROM_IMPL`

**Default value:**
- No (disabled)

Deprecated options and their replacements

- `CONFIG_A2DP_ENABLE` (**CONFIG_BT_A2DP_ENABLE**)
- `CONFIG_A2D_INITIAL_TRACE_LEVEL` (**CONFIG_BT_LOG_A2D_TRACE_LEVEL**)
  - `CONFIG_A2D_TRACE_LEVEL_NONE`
  - `CONFIG_A2D_TRACE_LEVEL_ERROR`
  - `CONFIG_A2D_TRACE_LEVEL_WARNING`
  - `CONFIG_A2D_TRACE_LEVEL_API`
  - `CONFIG_A2D_TRACE_LEVEL_EVENT`
  - `CONFIG_A2D_TRACE_LEVEL_DEBUG`
• CONFIG_A2D_TRACE_LEVEL_VERBOSE
  • CONFIG_ADC2_DISABLE_DAC (CONFIG_ADC_DISABLE_DAC)
  • CONFIG_APP_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_APPL_TRACE_LEVEL)
    – CONFIG_APPL_TRACE_LEVEL_NONE
    – CONFIG_APPL_TRACE_LEVEL_ERROR
    – CONFIG_APPL_TRACE_LEVEL_WARNING
    – CONFIG_APPL_TRACE_LEVEL_API
    – CONFIG_APPL_TRACE_LEVEL_EVENT
    – CONFIG_APPL_TRACE_LEVEL_DEBUG
    – CONFIG_APPL_TRACE_LEVEL_VERBOSE
  • 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_AVCT_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_AVCT_TRACE_LEVEL)
    – CONFIG_AVCT_TRACE_LEVEL_NONE
    – CONFIG_AVCT_TRACE_LEVEL_ERROR
    – CONFIG_AVCT_TRACE_LEVEL_WARNING
    – CONFIG_AVCT_TRACE_LEVEL_API
    – CONFIG_AVCT_TRACE_LEVEL_EVENT
    – CONFIG_AVCT_TRACE_LEVEL_DEBUG
    – CONFIG_AVCT_TRACE_LEVEL_VERBOSE
  • CONFIG_AVDT_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_AVDT_TRACE_LEVEL)
    – CONFIG_AVDT_TRACE_LEVEL_NONE
    – CONFIG_AVDT_TRACE_LEVEL_ERROR
    – CONFIG_AVDT_TRACE_LEVEL_WARNING
    – CONFIG_AVDT_TRACE_LEVEL_API
    – CONFIG_AVDT_TRACE_LEVEL_EVENT
    – CONFIG_AVDT_TRACE_LEVEL_DEBUG
    – CONFIG_AVDT_TRACE_LEVEL_VERBOSE
  • CONFIG_AVRC_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_AVRC_TRACE_LEVEL)
    – CONFIG_AVRC_TRACE_LEVEL_NONE
    – CONFIG_AVRC_TRACE_LEVEL_ERROR
    – CONFIG_AVRC_TRACE_LEVEL_WARNING
    – CONFIG_AVRC_TRACE_LEVEL_API
    – CONFIG_AVRC_TRACE_LEVEL_EVENT
    – CONFIG_AVRC_TRACE_LEVEL_DEBUG
    – CONFIG_AVRC_TRACE_LEVEL_VERBOSE
  • CONFIG_BLE_ACTIVE_SCAN_REPORT_ADV_SCAN_RSP_INDIVIDUALLY (CONFIG_BT_BLE_ACT_SCAN_REP_ADV_SCAN)
  • CONFIG_BLE_ESTABLISH_LINK_CONNECTION_TIMEOUT (CONFIG_BT_BLE_ESTAB_LINK_CONN_TOUT)
  • CONFIG_BLE_HOST_QUEUE_CONGESTION_CHECK (CONFIG_BT_BLE_HOST_QUEUE_CONG_CHECK)
  • CONFIG_BLE_MESH_GATT_PROXY (CONFIG_BT_MESH_GATT_PROXY_SERVER)
  • 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_BLUFI_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_BLUFI_TRACE_LEVEL)
    – CONFIG_BLUFI_TRACE_LEVEL_NONE
    – CONFIG_BLUFI_TRACE_LEVEL_ERROR
    – CONFIG_BLUFI_TRACE_LEVEL_WARNING
    – CONFIG_BLUFI_TRACE_LEVEL_API
    – CONFIG_BLUFI_TRACE_LEVEL_EVENT
    – CONFIG_BLUFI_TRACE_LEVEL_DEBUG
    – CONFIG_BLUFI_TRACE_LEVEL_VERBOSE
• **CONFIG_BNEP_INITIAL_TRACE_LEVEL** ([CONFIG_BT_LOG_BNEP_TRACE_LEVEL](#))
• **CONFIG_BROWNOUT_DET** ([CONFIG_ESP_BROWNOUT_DET](#))
• **CONFIG_BROWNOUT_DET_LVL_SEL** ([CONFIG_ESP_BROWNOUT_DET_LVL_SEL](#))
  - `CONFIG_BROWNOUT_DET_LVL_SEL_7`
  - `CONFIG_BROWNOUT_DET_LVL_SEL_6`
  - `CONFIG_BROWNOUT_DET_LVL_SEL_5`
  - `CONFIG_BROWNOUT_DET_LVL_SEL_4`
  - `CONFIG_BROWNOUT_DET_LVL_SEL_3`
  - `CONFIG_BROWNOUT_DET_LVL_SEL_2`
  - `CONFIG_BROWNOUT_DET_LVL_SEL_1`
• **CONFIG_BTC_INITIAL_TRACE_LEVEL** ([CONFIG_BT_LOG_BTC_TRACE_LEVEL](#))
  - `CONFIG_BTC_TRACE_LEVEL_NONE`
  - `CONFIG_BTC_TRACE_LEVEL_ERROR`
  - `CONFIG_BTC_TRACE_LEVEL_WARNING`
  - `CONFIG_BTC_TRACE_LEVEL_API`
  - `CONFIG_BTC_TRACE_LEVEL_EVENT`
  - `CONFIG_BTC_TRACE_LEVEL_DEBUG`
  - `CONFIG_BTC_TRACE_LEVEL_VERBOSE`
• **CONFIG_BTC_TASK_STACK_SIZE** ([CONFIG_BT_BTU_TASK_STACK_SIZE](#))
• **CONFIG_BTH_LOG_SDP_INITIAL_TRACE_LEVEL** ([CONFIG_BT_LOG_SDP_TRACE_LEVEL](#))
  - `CONFIG_SDP_TRACE_LEVEL_NONE`
  - `CONFIG_SDP_TRACE_LEVEL_ERROR`
  - `CONFIG_SDP_TRACE_LEVEL_WARNING`
  - `CONFIG_SDP_TRACE_LEVEL_API`
  - `CONFIG_SDP_TRACE_LEVEL_EVENT`
  - `CONFIG_SDP_TRACE_LEVEL_DEBUG`
  - `CONFIG_SDP_TRACE_LEVEL_VERBOSE`
• **CONFIG_BTM_INITIAL_TRACE_LEVEL** ([CONFIG_BT_LOG_BTM_TRACE_LEVEL](#))
  - `CONFIG_BTM_TRACE_LEVEL_NONE`
  - `CONFIG_BTM_TRACE_LEVEL_ERROR`
  - `CONFIG_BTM_TRACE_LEVEL_WARNING`
  - `CONFIG_BTM_TRACE_LEVEL_API`
  - `CONFIG_BTM_TRACE_LEVEL_EVENT`
  - `CONFIG_BTM_TRACE_LEVEL_DEBUG`
  - `CONFIG_BTM_TRACE_LEVEL_VERBOSE`
• **CONFIG_BTU_TASK_STACK_SIZE** ([CONFIG_BT_BTU_TASK_STACK_SIZE](#))
• **CONFIG_BT_NIMBLE_ACL_BUF_COUNT** ([CONFIG_BT_NIMBLE_TRANSPORT_ACL_FROM_LL_COUNT](#))
• **CONFIG_BT_NIMBLE_ACL_BUF_SIZE** ([CONFIG_BT_NIMBLE_TRANSPORT_ACL_SIZE](#))
• **CONFIG_BT_NIMBLE_HCI_EVT_BUF_SIZE** ([CONFIG_BT_NIMBLE_TRANSPORT_EVT_SIZE](#))
• **CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT** ([CONFIG_BT_NIMBLE_TRANSPORT_EVT_COUNT](#))
• **CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT** ([CONFIG_BT_NIMBLE_TRANSPORT_EVT_DISCARD_COUNT](#))
• **CONFIG_BT_NIMBLE_MSYS1_BLOCK_COUNT** ([CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT](#))
• **CONFIG_BT_NIMBLE_TASK_STACK_SIZE** ([CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE](#))
• **CONFIG_CLASSIC_BT_ENABLED** ([CONFIG_BT_CLASSIC_ENABLED](#))
• **CONFIG_CONSOLE_UART** ([CONFIG_ESP_CONSOLE_UART](#))
  - `CONFIG_CONSOLE_UART_DEFAULT`
  - `CONFIG_CONSOLE_UART_CUSTOM`
  - `CONFIG_CONSOLE_UART_NONE`, `[CONFIG_ESP_CONSOLE_UART_NONE](#)`
• **CONFIG_CONSOLE_UART_BAUDRATE** ([CONFIG_ESP_CONSOLE_UART_BAUDRATE](#))
• **CONFIG_CONSOLE_UART_NUM** ([CONFIG_ESP_CONSOLE_UART_NUM](#))

---

Espressif Systems 1217 Release master
Submit Document Feedback
Chapter 2. API Reference

- CONFIG_CONSOLE_UART_CUSTOM_NUM_0
- CONFIG_CONSOLE_UART_CUSTOM_NUM_1
  - 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_ENABLE_STATIC_TASK_CLEAN_UP_HOOK (CONFIG_FREERTOS_ENABLE_STATIC_TASK_CLEAN_UP_HOOK)
- 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_TRAX_THRESH)
- CONFIG_ESP32_CORE_DUMP_CODEC (CONFIG_ESP_COREDUMP_CODEC)
  - CONFIG_ESP32_CORE_DUMP_CODEC_INFO
  - CONFIG_ESP32_CORE_DUMP_CODEC_DISABLE
- CONFIG_ESP32_CORE_DUMP_MAX_TASKS_NUM (CONFIG_ESP_COREDUMP_MAX_TASKS_NUM)
- CONFIG_ESP32_CORE_DUMP_STACK_SIZE (CONFIG_ESP_COREDUMP_STACK_SIZE)
- CONFIG_ESP32_CORE_DUMP_UART_DELAY (CONFIG_ESP_COREDUMP_UART_DELAY)
- CONFIG_ESP32_DEBUG_STUBS_ENABLE (CONFIG_ESP_DEBUG_STUBS_ENABLE)
- CONFIG_ESP32_GCOV_ENABLE (CONFIG_APPTRACE_GCOV_ENABLE)
- CONFIG_ESP32_PHY_CALIBRATION_AND_DATA_STORAGE (CONFIG_ESP_PHY_CALIBRATION_AND_DATA_STORAGE)
- CONFIG_ESP32_PHY_DEFAULT_INIT_IF_INVALID (CONFIG_ESP_PHY_DEFAULT_INIT_IF_INVALID)
- CONFIG_ESP32_PHY_INIT_DATA_ERROR (CONFIG_ESP_PHY_INIT_DATA_ERROR)
- CONFIG_ESP32_PHY_INIT_DATA_IN_PARTITION (CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION)
- CONFIG_ESP32_PHY_MAC_BB_PD (CONFIG_ESP_PHY_MAC_BB_PD)
- CONFIG_ESP32_PHY_MAX_WIFI_TX_POWER (CONFIG_ESP_PHY_MAX_WIFI_TX_POWER)
- CONFIG_ESP32_PTHREAD_STACK_MIN (CONFIG_PTHREAD_STACK_MIN)
- CONFIG_ESP32_PTHREAD_TASK_CORE_DEFAULT (CONFIG_PTHREAD_TASK_CORE_DEFAULT)
  - CONFIG_ESP32_DEFAULT_PTHREAD_CORE_NO_AFFINITY
  - CONFIG_ESP32_DEFAULT_PTHREAD_CORE_0
  - CONFIG_ESP32_DEFAULT_PTHREAD_CORE_1
- 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_XTAL_BOOTSTRAP_CYCLES (CONFIG_ESP_SYSTEM_RTC_EXT_XTAL_BOOTSTRAP_CYCLES)
- CONFIG_ESP32_SUPPORT_MULTIPLE_PHY_INIT_DATA_BIN (CONFIG_ESP_PHY_MULTIPLE_INIT_DATA_BIN)
- CONFIG_ESP32_WIFI_AMPDU_RX_ENABLED (CONFIG_ESP_WIFI_AMPDU_RX_ENABLED)
- CONFIG_ESP32_WIFI_AMPDU_TX_ENABLED (CONFIG_ESP_WIFI_AMPDU_TX_ENABLED)
- CONFIG_ESP32_WIFI_AMSDU_TX_ENABLED (CONFIG_ESP_WIFI_AMSDU_TX_ENABLED)
- CONFIG_ESP32_WIFI_CACHE_TX_BUFFER_NUM (CONFIG_ESP_WIFI_CACHE_TX_BUFFER_NUM)
- CONFIG_ESP32_WIFI_CSI_ENABLED (CONFIG_ESP_WIFI_CSI_ENABLED)
- CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM (CONFIG_ESP_WIFI_DYNAMIC_RX_BUFFER_NUM)
- CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER_NUM (CONFIG_ESP_WIFI_DYNAMIC_TX_BUFFER_NUM)
- CONFIG_ESP32_WIFI_ENABLE_WPA3_OWE_STA (CONFIG_ESP_WIFI_ENABLE_WPA3_OWE_STA)
- CONFIG_ESP32_WIFI_ENABLE_WPA3_SAE (CONFIG_ESP_WIFI_ENABLE_WPA3_SAE)
- CONFIG_ESP32_WIFI_IRAM_OPT (CONFIG_ESP_WIFI_IRAM_OPT)
- CONFIG_ESP32_WIFI_MGMT_SBUF_NUM (CONFIG_ESP_WIFI_MGMT_SBUF_NUM)
- CONFIG_ESP32_WIFI_NVSD_ENABLE (CONFIG_ESP_WIFI_NVSD_ENABLE)
- CONFIG_ESP32_WIFI_RX_BA_WIN (CONFIG_ESP_WIFI_RX_BA_WIN)
- CONFIG_ESP32_WIFI_RX_IRAM_OPT (CONFIG_ESP_WIFI_RX_IRAM_OPT)
- CONFIG_ESP32_WIFI_SOFTAP_BEACON_MAX_LEN (CONFIG_ESP_WIFI_SOFTAP_BEACON_MAX_LEN)
- CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM (CONFIG_ESP_WIFI_STATIC_RX_BUFFER_NUM)
- CONFIG_ESP32_WIFI_STATIC_TX_BUFFER_NUM (CONFIG_ESP_WIFI_STATIC_TX_BUFFER_NUM)
Chapter 2. API Reference

- **CONFIG_ESP32_WIFI_SW_COEXIST_ENABLE**
- **CONFIG_ESP32_WIFI_TASK_CORE_ID**
  - CONFIG_ESP32_WIFI_TASK_Pinned_TO_CORE_0
  - CONFIG_ESP32_WIFI_TASK_Pinned_TO_CORE_1
- **CONFIG_ESP32_WIFI_TX_BA_WIN**
- **CONFIG_ESP32_WIFI_TX_BUFFER**
  - CONFIG_ESP32_WIFI_STATIC_TX_BUFFER
  - CONFIG_ESP32_WIFI_DYNAMIC_TX_BUFFER
- **CONFIG_ESP_GRATUITOUS_ARP**
- **CONFIG_ESP_SYSTEM_PD_FLASH**
- **CONFIG_ESP_SYSTEM_PM_POWER_DOWN_CPU**
- **CONFIG_ESP_TASK_WDT**
- **CONFIG_ESP_WIFI_EXTERNAL_COEXIST_ENABLE**
- **CONFIG_ESP_WIFI_SW_COEXIST_ENABLE**
- **CONFIG_EVENT_LOOP_PROFILING**
- **CONFIG_EXTERNAL_COEX_ENABLE**
- **CONFIG_FLASH_ENCRYPTION_ENABLED**
- **CONFIG_FLASH_ENCRYPTION_UART_BOOTLOADER_ALLOW_CACHE**
- **CONFIG_GAP_INITIAL_TRACE_LEVEL**
  - CONFIG_GAP_TRACE_LEVEL_NONE
  - CONFIG_GAP_TRACE_LEVEL_ERROR
  - CONFIG_GAP_TRACE_LEVEL_WARNING
  - CONFIG_GAP_TRACE_LEVEL_API
  - CONFIG_GAP_TRACE_LEVEL_EVENT
  - CONFIG_GAP_TRACE_LEVEL_DEBUG
  - CONFIG_GAP_TRACE_LEVEL_VERBOSE
- **CONFIG_GARP_TMR_INTERVAL**
- **CONFIG_GATTC_CACHE_NVS_FLASH**
- **CONFIG_GATTC_ENABLE**
- **CONFIG_GATT_INITIAL_TRACE_LEVEL**
  - CONFIG_GATT_TRACE_LEVEL_NONE
  - CONFIG_GATT_TRACE_LEVEL_ERROR
  - CONFIG_GATT_TRACE_LEVEL_WARNING
  - CONFIG_GATT_TRACE_LEVEL_API
  - CONFIG_GATT_TRACE_LEVEL_EVENT
  - CONFIG_GATT_TRACE_LEVEL_DEBUG
  - CONFIG_GATT_TRACE_LEVEL_VERBOSE
- **CONFIG_GATTS_SEND_SERVICE_CHANGE_MODE**
  - CONFIG_GATTS_SEND_SERVICE_CHANGE_MANUAL
  - CONFIG_GATTS_SEND_SERVICE_CHANGE_AUTO
- **CONFIG_GATT_INITIAL_TRACE_LEVEL**
  - CONFIG_GATT_TRACE_LEVEL_NONE
  - CONFIG_GATT_TRACE_LEVEL_ERROR
  - CONFIG_GATT_TRACE_LEVEL_WARNING
  - CONFIG_GATT_TRACE_LEVEL_API
  - CONFIG_GATT_TRACE_LEVEL_EVENT
  - CONFIG_GATT_TRACE_LEVEL_DEBUG
  - CONFIG_GATT_TRACE_LEVEL_VERBOSE
- **CONFIG_HCI_INITIAL_TRACE_LEVEL**
  - CONFIG_HCI_TRACE_LEVEL_NONE
  - CONFIG_HCI_TRACE_LEVEL_ERROR
  - CONFIG_HCI_TRACE_LEVEL_WARNING
  - CONFIG_HCI_TRACE_LEVEL_API
  - CONFIG_HCI_TRACE_LEVEL_EVENT
  - CONFIG_HCI_TRACE_LEVEL_DEBUG
  - CONFIG_HCI_TRACE_LEVEL_VERBOSE
- **CONFIG_HFP_AG_ENABLE**
- **CONFIG_HFP_AUDIO_DATA_PATH**
  - CONFIG_HFP_AUDIO_DATA_PATH_PCM
Chapter 2. API Reference

- CONFIG_HFP_AUDIO_DATA_PATH_HCI
- CONFIG_HFP_CLIENT_ENABLE (CONFIG_BT_HFP_CLIENT_ENABLE)
- CONFIG_HFP_ENABLE (CONFIG_BT_HFP_ENABLE)
- CONFIG_HID_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_HID_TRACE_LEVEL)
  - CONFIG_HID_TRACE_LEVEL_NONE
  - CONFIG_HID_TRACE_LEVEL_ERROR
  - CONFIG_HID_TRACE_LEVEL_WARNING
  - CONFIG_HID_TRACE_LEVEL_API
  - CONFIG_HID_TRACE_LEVEL_EVENT
  - CONFIG_HID_TRACE_LEVEL_DEBUG
  - CONFIG_HID_TRACE_LEVEL_VERBOSE
- 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_L2CAP_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_L2CAP_TRACE_LEVEL)
  - CONFIG_L2CAP_TRACE_LEVEL_NONE
  - CONFIG_L2CAP_TRACE_LEVEL_ERROR
  - CONFIG_L2CAP_TRACE_LEVEL_WARNING
  - CONFIG_L2CAP_TRACE_LEVEL_API
  - CONFIG_L2CAP_TRACE_LEVEL_EVENT
  - CONFIG_L2CAP_TRACE_LEVEL_DEBUG
  - CONFIG_L2CAP_TRACE_LEVEL_VERBOSE
- CONFIG_L2_TO_L3_COPY (CONFIG_LWIP_L2_TO_L3_COPY)
- CONFIG_LOG_BOOTLOADER_LEVEL (CONFIG_BOOTLOADER_LOG_LEVEL)
  - CONFIG_LOG_BOOTLOADER_LEVEL_NONE
  - CONFIG_LOG_BOOTLOADER_LEVEL_ERROR
  - CONFIG_LOG_BOOTLOADER_LEVEL_WARNING
  - CONFIG_LOG_BOOTLOADER_LEVEL_INFO
  - CONFIG_LOG_BOOTLOADER_LEVEL_DEBUG
  - CONFIG_LOG_BOOTLOADER_LEVEL_VERBOSE
- CONFIG_MAC_BB_PD (CONFIG_ESP_PHY_MAC_BB_PD)
- CONFIG_MAIN_TASK_STACK_SIZE (CONFIG_ESP_MAIN_TASK_STACK_SIZE)
- CONFIG_MCA_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_MCA_TRACE_LEVEL)
  - CONFIG_MCA_TRACE_LEVEL_NONE
  - CONFIG_MCA_TRACE_LEVEL_ERROR
  - CONFIG_MCA_TRACE_LEVEL_WARNING
  - CONFIG_MCA_TRACE_LEVEL_API
  - CONFIG_MCA_TRACE_LEVEL_EVENT
  - CONFIG_MCA_TRACE_LEVEL_DEBUG
  - CONFIG_MCA_TRACE_LEVEL_VERBOSE
- CONFIG_MCPWM_ISR_IN_IRAM (CONFIG_MCPWM_ISR_IRAM_SAFE)
- CONFIG_NIMBLE_ATT_PREFERRED_MTU (CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU)
- CONFIG_NIMBLE_CRYPTO_STACK_MBEDTLS (CONFIG_BT_NIMBLE_CRYPTO_STACK_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_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_PROXY (CONFIG_BT_NIMBLE_MESH_PROXY)
  • CONFIG_NIMBLE_MESH_RELAY (CONFIG_BT_NIMBLE_MESH_RELAY)
  • 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_HOST_TASK_STACK_SIZE)
  • CONFIG_NO_BLOBS (CONFIG_APP_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_OPTIMIZATION_LEVEL_DEBUG, CONFIG_COMPILER_OPTIMIZATION_LEVEL_DEBUG,
    CONFIG_COMPILER_OPTIMIZATION_DEFAULT
    • CONFIG_OPTIMIZATION_LEVEL_RELEASE, CONFIG_COMPILER_OPTIMIZATION_LEVEL_RELEASE
  • CONFIG_OSI_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_OSI_TRACE_LEVEL)
    • CONFIG_OSI_TRACE_LEVEL_NONE
    • CONFIG_OSI_TRACE_LEVEL_ERROR
    • CONFIG_OSI_TRACE_LEVEL_WARNING
    • CONFIG_OSI_TRACE_LEVEL_API
    • CONFIG_OSI_TRACE_LEVEL_EVENT
    • CONFIG_OSI_TRACE_LEVEL_DEBUG
    • CONFIG_OSI_TRACE_LEVEL_VERBOSE
  • CONFIG_OSA_ALLOW_HTTP (CONFIG_ESP_HTTPS_OSA_ALLOW_HTTP)
  • CONFIG_PAN_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_PAN_TRACE_LEVEL)
    • CONFIG_PAN_TRACE_LEVEL_NONE
    • CONFIG_PAN_TRACE_LEVEL_ERROR
    • CONFIG_PAN_TRACE_LEVEL_WARNING
    • CONFIG_PAN_TRACE_LEVEL_API
    • CONFIG_PAN_TRACE_LEVEL_EVENT
    • CONFIG_PAN_TRACE_LEVEL_DEBUG
    • CONFIG_PAN_TRACE_LEVEL_VERBOSE
  • 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_REDUCE_PHY_TX_POWER (CONFIG_ESP_PHY_REDUCE_TX_POWER)
• CONFIG_RFCOMM_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_RFCOMM_TRACE_LEVEL)
  - CONFIG_RFCOMM_TRACE_LEVEL_NONE
  - CONFIG_RFCOMM_TRACE_LEVEL_ERROR
  - CONFIG_RFCOMM_TRACE_LEVEL_WARNING
  - CONFIG_RFCOMM_TRACE_LEVEL_API
  - CONFIG_RFCOMM_TRACE_LEVEL_EVENT
  - CONFIG_RFCOMM_TRACE_LEVEL_DEBUG
  - CONFIG_RFCOMM_TRACE_LEVEL_VERBOSE
• CONFIG_SEMIHOSTFS_MAX_MOUNT_POINTS (CONFIG_VFS_SEMIHOSTFS_MAX_MOUNT_POINTS)
• CONFIG_SMP_INITIAL_TRACE_LEVEL (CONFIG_BT_LOG_SMP_TRACE_LEVEL)
  - CONFIG_SMP_TRACE_LEVEL_NONE
  - CONFIG_SMP_TRACE_LEVEL_ERROR
  - CONFIG_SMP_TRACE_LEVEL_WARNING
  - CONFIG_SMP_TRACE_LEVEL_API
  - CONFIG_SMP_TRACE_LEVEL_EVENT
  - CONFIG_SMP_TRACE_LEVEL_DEBUG
  - CONFIG_SMP_TRACE_LEVEL_VERBOSE
• 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_TERMIOS (CONFIG_VFS_SUPPORT_TERMIOS)
• CONFIG_SUPPRESS_SELECT_DEBUG_OUTPUT (CONFIG_VFS_SUPPRESS_SELECT_DEBUG_OUTPUT)
• CONFIG_SYSVIEW_BUF_WAIT_TMO (CONFIG_APPTRACE_SV_BUF_WAIT_TMO)
• CONFIG_SYSVIEW_ENABLE (CONFIG_APPTRACE_SV_ENABLE)
• CONFIG_SYSVIEW_EVT_ISR_ENTER_ENABLE (CONFIG_APPTRACE_SV_EVT_ISR_ENTER_ENABLE)
• CONFIG_SYSVIEW_EVT_ISR_EXIT_ENABLE (CONFIG_APPTRACE_SV_EVT_ISR_EXIT_ENABLE)
• CONFIG_SYSVIEW_EVT_ISR_TO_SCHEDULER_ENABLE (CONFIG_APPTRACE_SV_EVT_ISR_TO_SCHED_ENABLE)
• CONFIG_SYSVIEW_EVT_TASK_CREATE_ENABLE (CONFIG_APPTRACE_SV_EVT_TASK_CREATE_ENABLE)
• CONFIG_SYSVIEW_EVT_TASK_START_EXEC_ENABLE (CONFIG_APPTRACE_SV_EVT_TASK_START_EXEC_ENABLE)
• CONFIG_SYSVIEW_EVT_TASK_START_READY_ENABLE (CONFIG_APPTRACE_SV_EVT_TASK_START_READY_ENABLE)
• CONFIG_SYSVIEW_EVT_TASK_STOP_EXEC_ENABLE (CONFIG_APPTRACE_SV_EVT_TASK_STOP_EXEC_ENABLE)
• CONFIG_SYSVIEW_EVT_TASK_STOP_READY_ENABLE (CONFIG_APPTRACE_SV_EVT_TASK_STOP_READY_ENABLE)
• CONFIG_SYSVIEW_EVT_TASK_TERMINATE_ENABLE (CONFIG_APPTRACE_SV_EVT_TASK_TERMINATE_ENABLE)
• CONFIG_SYSVIEW_EVT_TIMER_ENTER_ENABLE (CONFIG_APPTRACE_SV_EVT_TIMER_ENTER_ENABLE)
• CONFIG_SYSVIEW_EVT_TIMER_EXIT_ENABLE (CONFIG_APPTRACE_SV_EVT_TIMER_EXIT_ENABLE)
• CONFIG_SYSVIEW_MAX_TASKS (CONFIG_APPTRACE_SV_MAX_TASKS)
• CONFIG_SYSVIEW_TS_SOURCE (CONFIG_APPTRACE_SV_TS_SOURCE)
  - CONFIG_SYSVIEW_TS_SOURCE_CCOUNT
  - CONFIG_SYSVIEW_TS_SOURCE_ESP_TIMER
2.7 Provisioning API

2.7.1 Protocol Communication

Overview

The Protocol Communication (protocomm) component manages secure sessions and provides the framework for multiple transports. The application can also use the 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 the application level
  - `protocomm_security0` (no security)
  - `protocomm_security1` (Curve25519 key exchange + AES-CTR encryption/decryption)
  - `protocomm_security2` (SRP6a-based key exchange + AES-GCM encryption/decryption)
- Proof-of-possession (support with `protocomm_security1` only)
- Salt and Verifier (support with `protocomm_security2` only)

Protocomm internally uses protobuf (protocol buffers) for secure session establishment. Users can choose to implement their own security (even without using protobuf). Protocomm can also be used without any security layer.

Protocomm provides the framework for various transports:

- Wi-Fi (SoftAP + HTTPD)
- 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 for `protocomm_security1` and `protocomm_security2`, the client still needs to establish sessions by performing the two-way handshake. See **Unified Provisioning** for more details about the secure handshake logic.

### Enabling Protocomm Security Version

The protocomm component provides a project configuration menu to enable/disable support of respective security versions. The respective configuration options are as follows:

- Support `protocomm_security0`, with no security: `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0`, this option is enabled by default.
- Support `protocomm_security1` with Curve25519 key exchange + AES-CTR encryption/decryption: `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1`, this option is enabled by default.
- Support `protocomm_security2` with SRP6a-based key exchange + AES-GCM encryption/decryption: `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2`.

**Note:** Enabling multiple security versions at once offers the ability to control them dynamically but also increases the firmware size.

### SoftAP + HTTP Transport Example with Security 2

For sample usage, see `wifi_provisioning/src/scheme_softap.c`.

```c
/* The endpoint handler to be registered with protocomm. This simply echoes back the received data. */
void 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 the data length updated. */
    *outbuf = malloc(inlen); /* This is to be deallocated outside. */
    memcpy(*outbuf, inbuf, inlen);

    /* Private data that was passed at the time of endpoint creation. */

    (continues on next page)
```
(continued from previous page)

```c
uint32_t *priv = (uint32_t *) priv_data;
if (priv) {
    printf("Private data : ", *priv);
}

return ESP_OK;
}

static const char sec2_salt[] = {0x0f7, 0x5f5, 0xe2e, 0xbeb, 0xba0, 0x7c0, 0x810, 0xcd0};
static const char sec2_verifier[] = {0xbff, 0x860, 0xce0, 0x630, 0x8a0, 0xbb8, 0x7e0};
```

Chapter 2. API Reference

(continued from previous page)

```c
/* The example function for launching a protocomm instance over HTTP. */
protocomm_t *start_pc() {
    protocomm_t *pc = protocomm_new();

    /* Config for protocomm_httpd_start(). */
    protocomm_httpd_config_t pc_config = {
        .data = {
            .config = PROTOCOMM_HTTPD_DEFAULT_CONFIG
        }
    }
```
/* Start the protocomm server on top of HTTP. */
protocomm_httpd_start(pc, &pc_config);

/* Create Security2 params object from salt and verifier. It must be valid throughout the scope of protocomm endpoint. This does not need to be static, i.e., could be dynamically allocated and freed at the time of endpoint removal. */
const static protocomm_security2_params_t sec2_params = {
    .salt = (const uint8_t *)salt,
    .salt_len = sizeof(salt),
    .verifier = (const uint8_t *) verifier,
    .verifier_len = sizeof(verifier),
};

/* Set security for communication at the 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_security2, &sec2_params);

/* Private data passed to the endpoint must be valid throughout the scope of protocomm endpoint. This need not be static, i.e., 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 the 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;

/* The example function for stopping a protocomm instance. */
void stop_pc(protocomm_t *pc)
{
    /* Remove the endpoint identified by its unique name. */
    protocomm_remove_endpoint(pc, "echo_req_endpoint");

    /* Remove the security endpoint identified by its name. */
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop the HTTP server. */
    protocomm_httpd_stop(pc);

    /* Delete, namely deallocate the protocomm instance. */
    protocomm_delete(pc);
}

SoftAP + HTTP Transport Example with Security 1

For sample usage, see wifi_provisioning/src/scheme_softap.c.

/* The 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 the data length updated. */
    *outbuf = malloc(inlen); /* This is to 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;
}

/* The 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 = {
        .data = {
            .config = PROTOCOMM_HTTPD_DEFAULT_CONFIG()
        }
    };

    /* Start the protocomm server on top of HTTP. */
    protocomm_httpd_start(pc, &pc_config);

    /* Create security1 params object from pop_string. It must be valid throughout...
    the scope of protocomm endpoint. This need not be static, i.e., could be...
    dynamically allocated and freed at the time of endpoint removal. */
    const static protocomm_security1_params_t secl_params = {
        .data = (const uint8_t *) strdup(pop_string),
        .len = strlen(pop_string)
    };

    /* Set security for communication at the 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, &secl___
params);

    /* Private data passed to the endpoint must be valid throughout the scope of...
    protocomm endpoint. This need not be static, i.e., 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 the 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;
}

/* The example function for stopping a protocomm instance. */
void stop_pc(protocomm_t *pc)
{
    /* Remove the endpoint identified by its unique name. */
    protocomm_remove_endpoint(pc, "echo_req_endpoint");

    /* Remove the security endpoint identified by its name. */
    protocomm_unset_security(pc, "security_endpoint");

    /* Stop the HTTP server. */
    protocomm_httpd_stop(pc);

    /* Delete, namely deallocate the protocomm instance. */
    protocomm_delete(pc);
}

API Reference

Header File

- components/protocomm/include/common/protocomm.h

This header file can be included with:

```
#include "protocomm.h"
```

- This header file is a part of the API provided by the protocomm component. To declare that your component depends on protocomm, add the following to your CMakeLists.txt:

```
REQUIRES protocomm
```

or

```
PRIV_REQUIRES protocomm
```

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.

**Returns**

- `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**

- `pc` — [in] 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 passed 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.

**Parameters**
- `pc` - [in] Pointer to the protocomm instance
- `ep_name` – [in] Endpoint identifier (name) string
- `h` – [in] Endpoint handler function
- `priv_data` – [in] Pointer to private data to be passed as a parameter to the handler function on call. Pass NULL if not needed.

**Returns**
- `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

```c
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.

**Parameters**
- `pc` - [in] Pointer to the protocomm instance
- `ep_name` – [in] Endpoint identifier (name) string

**Returns**
- `ESP_OK`: Success
- `ESP_ERR_NOT_FOUND`: Endpoint with specified name doesn’t exist
- `ESP_ERR_INVALID_ARG`: Null instance/name arguments

```c
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()`.

**Parameters**
- `pc` - [in] Pointer to the protocomm instance
- `session_id` – [in] Unique ID for a communication session

**Returns**
- `ESP_OK`: Request handled successfully
- `ESP_ERR_NO_MEM`: Error allocating internal resource
- `ESP_ERR_INVALID_ARG`: Null instance/name arguments
**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()`.

**Parameters**
- `pc` – [in] Pointer to the protocomm instance
- `session_id` – [in] Unique ID for a communication session

**Returns**
- ESP_OK: Request handled successfully
- ESP_ERR_INVALID_ARG: Null instance/name arguments

**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.

**Parameters**
- `pc` – [in] Pointer to the protocomm instance
- `ep_name` – [in] Endpoint identifier(name) string
- `session_id` – [in] Unique ID for a communication session
- `inbuf` – [in] Input buffer contains input request data which is to be processed by the registered handler
- `inlen` – [in] Length of the input buffer
- `outbuf` – [out] Pointer to internally allocated output buffer, where the resulting response data output from the registered handler is to be stored
- `outlen` – [out] Buffer length of the allocated output buffer

**Returns**
- 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

**esp_err_t protocomm_set_security(protocomm_t *pc, const char* ep_name, const protocomm_security_t *sec, const void* sec_params)**

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`, `protocomm_security1`, and `protocomm_security2` are readily available.

**Parameters**
**Chapter 2. API Reference**

- **pc** - [in] Pointer to the protocomm instance
- **ep_name** - [in] Endpoint identifier(name) string
- **sec** - [in] Pointer to endpoint security instance
- **sec_params** - [in] Pointer to security params (NULL if not needed) The pointer should contain the security params struct of appropriate security version. For protocomm security version 1 and 2 sec_params should contain pointer to struct of type protocomm_security1_params_t and protocomm_security2_params_t respectively. The contents of this pointer must be valid till the security session has been running and is not closed.

**Returns**
- 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

**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.

**Parameters**
- **pc** - [in] Pointer to the protocomm instance
- **ep_name** - [in] Endpoint identifier(name) string

**Returns**
- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn’t exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

**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().

**Parameters**
- **pc** - [in] Pointer to the protocomm instance
- **ep_name** - [in] Endpoint identifier(name) string
- **version** - [in] Version identifier(name) string

**Returns**
- 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

**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.

**Parameters**
- **pc** - [in] Pointer to the protocomm instance
- **ep_name** - [in] Endpoint identifier(name) string

**Returns**
- ESP_OK : Success
Chapter 2. API Reference

- ESP_OK : Success
- ESP_ERR_NOT_FOUND : Endpoint with specified name doesn’t exist
- ESP_ERR_INVALID_ARG : Null instance/name arguments

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

- components/protocomm/include/security/protocomm_security.h
- This header file can be included with:

```
#include "protocomm_security.h"
```
- This header file is a part of the API provided by the protocomm component. To declare that your component depends on protocomm, add the following to your CMakeLists.txt:

```
REQUIRES protocomm
```

or

```
PRIV_REQUIRES protocomm
```

Structures

struct protocomm_security1_params

Protocomm Security 1 parameters: Proof Of Possession.

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_security2_params

Protocomm Security 2 parameters: Salt and Verifier.

Public Members

const char* salt

Pointer to the buffer containing the salt
**Chapter 2. API Reference**

```c
uint16_t salt_len
    Length (in bytes) of the salt

const char *verifier
    Pointer to the buffer containing the verifier

uint16_t verifier_len
    Length (in bytes) of the verifier
```

**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**

```c
int ver
    Unique version number of security implementation

esp_err_t(*init)(protocomm_security_handle_t *handle)
    Function for initializing/allocationg 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 void *sec_params,
    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**

```c
typedef struct protocomm_security1_params protocomm_security1_params_t
    Protocomm Security 1 parameters: Proof Of Possession.
```
Chapter 2. API Reference

typedef protocomm_security1_params_t protocomm_security_pop_t

typedef struct protocomm_security2_params protocomm_security2_params_t

Protocomm Security 2 parameters: Salt and Verifier.

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

Enumerations

enum protocomm_security_session_event_t

Events generated by the protocomm security layer.

These events are generated while establishing secured session.

Values:

enumerator PROTOCOMM_SECURITY_SESSION_SETUP_OK

Secured session established successfully

enumerator PROTOCOMM_SECURITY_SESSION_INVALID_SECURITY_PARAMS

Received invalid (NULL) security parameters (username / client public-key)

enumerator PROTOCOMM_SECURITY_SESSION_CREDENTIALS_MISMATCH

Received incorrect credentials (username / PoP)

Header File

• components/protocomm/include/security/protocomm_security0.h

This header file can be included with:

```c
#include "protocomm_security0.h"
```

• This header file is a part of the API provided by the protocomm component. To declare that your component depends on protocomm, add the following to your CMakeLists.txt:

```cmake
REQUIRES protocomm
```

or

```cmake
PRIV_REQUIRES protocomm
```

Header File

• components/protocomm/include/security/protocomm_security1.h

• This header file can be included with:
Chapter 2. API Reference

```c
#include "protocomm_security1.h"
```

- This header file is a part of the API provided by the `protocomm` component. To declare that your component depends on `protocomm`, add the following to your CMakeLists.txt:

```cmake
REQUIRES protocomm
```

- Or

```cmake
PRIV_REQUIRES protocomm
```

**Header File**

- `components/protocomm/include/transports/protocomm_httpd.h`

  - This header file can be included with:

    ```c
    #include "protocomm_httpd.h"
    ```

  - This header file is a part of the API provided by the `protocomm` component. To declare that your component depends on `protocomm`, add the following to your CMakeLists.txt:

    ```cmake
    REQUIRES protocomm
    ```

    - Or

    ```cmake
    PRIV_REQUIRES protocomm
    ```

**Functions**

```c
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.

**Parameters**

- `pc` - [in] Protocomm instance pointer obtained from `protocomm_new()`
- `config` - [in] Pointer to config structure for initializing HTTP server

**Returns**

- `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

```c
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.

**Parameters**

- `pc` - [in] Same protocomm instance that was passed to `protocomm_httpd_start()`

**Returns**

- `ESP_OK` : Success
- `ESP_ERR_INVALID_ARG` : Null / incorrect protocomm instance pointer
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

PROTOCOL_HTTPD_DEFAULT_CONFIG()
Chapter 2. API Reference

Header File

- components/protocomm/include/transports/protocomm_ble.h
- This header file can be included with:

```
#include "protocomm_ble.h"
```

- This header file is a part of the API provided by the protocomm component. To declare that your component depends on protocomm, add the following to your CMakeLists.txt:

```
REQUIRES protocomm
```

or

```
PRIV_REQUIRES protocomm
```

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.

**Parameters**

- pc - [in] Protocomm instance pointer obtained from protocomm_new()
- config - [in] Pointer to config structure for initializing BLE

**Returns**

- 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

**esp_err_t protocomm_ble_stop(protocomm_t *pc)**

Stop Bluetooth Low Energy based transport layer for provisioning.

Stops 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.

**Parameters**

- pc - [in] Same protocomm instance that was passed to protocomm_ble_start()

**Returns**

- ESP_OK : Success
- ESP_FAIL : Simple BLE stop error
- ESP_ERR_INVALID_ARG : Null / incorrect protocomm instance

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
Chapter 2. API Reference

uint16_t uuid

UUID to be assigned to the BLE characteristic which is mapped to the handler

struct protocomm_ble_event_t

Structure for BLE events in Protocomm.

Public Members

uint16_t evt_type

This field indicates the type of BLE event that occurred.

uint16_t conn_handle

The handle of the relevant connection.

uint16_t conn_status

The status of the connection attempt; 0: the connection was successfully established. 0 BLE host error code: the connection attempt failed for the specified reason.

uint16_t disconnect_reason

Return code indicating the reason for the disconnect.

struct protocomm_ble_config

Config parameters for protocomm BLE service.

Public Members

char device_name[MAX_BLE_DEVNAME_LEN + 1]

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

uint8_t* manufacturer_data

BLE device manufacturer data pointer in advertisement

ssize_t manufacturer_data_len

BLE device manufacturer data length in advertisement

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

unsigned ble_bonding

BLE bonding
unsigned ble_sm_sc
    BLE security flag

unsigned ble_link_encryption
    BLE security flag

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

MAX_BLE_MANUFACTURER_DATA_LEN
    Theoretically, the limit for max manufacturer length remains same as BLE device name i.e. 31 bytes (max scan response size) - 1 byte (length) - 1 byte (type) = 29 bytes However, manufacturer data goes along with BLE device name in scan response. So, it is important to understand the actual length should be smaller than (29 - (BLE device name length) - 2).

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.

Enumerations

enum protocomm_transport_ble_event_t
    Events generated by BLE transport.
    These events are generated when the BLE transport is paired and disconnected.
    Values:

    enumerator PROTOCOL_TRANSPORT_BLE_CONNECTED

    enumerator PROTOCOL_TRANSPORT_BLE_DISCONNECTED

2.7.2 Unified Provisioning

Overview

The 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. The developers can choose to extend the device-side and phone-app
side implementations to accommodate their requirements for sending additional configuration data. The followings 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 is also left to the application to decide.

2. **Transport Flexibility**

The protocol can work on Wi-Fi (SoftAP + HTTP server) or on Bluetooth LE as a transport protocol. The framework provides an ability to add support for any other transport easily as long as command-response behavior can be supported on the transport.

3. **Security Scheme Flexibility**

It is 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 is WPA2 protected or Bluetooth LE with the “just-works” security. Or the applications may consider the transport to be insecure and may want application-level security. The unified provisioning framework allows the 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**

The unified provisioning subsystem supports Wi-Fi (SoftAP+HTTP server) and Bluetooth LE (GATT based) transport schemes. The following points need to be considered while selecting the best possible transport for provisioning:

1. The Bluetooth LE-based transport has the advantage of maintaining an intact communication channel between the device and the client during the provisioning, which ensures reliable provisioning feedback.
2. The Bluetooth LE-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 the user to go out of the phone app.
3. However, the Bluetooth LE transport consumes about 110 KB memory at runtime. If the product does not use the Bluetooth LE or Bluetooth functionality after provisioning is done, almost all the memory can be reclaimed and added into the heap.
4. The SoftAP-based transport is highly interoperable. However, there are a few considerations:
   - The device uses the same radio to host the SoftAP and also to connect to the configured AP. Since these could potentially be on different channels, it may cause connection status updates not to be reliably received by the phone.
   - The phone (client) has to disconnect from its current AP in order to connect to the SoftAP. The original network will get restored only when the provisioning process is complete, and the softAP is taken down.
5. The SoftAP transport does not require much additional memory for the Wi-Fi use cases.
6. The SoftAP-based provisioning requires the phone-app user to go to System Settings to connect to the Wi-Fi network hosted by the device in the iOS system. The discovery (scanning) as well as connection APIs are 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. The 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 have to be secured.
2. The client should authenticate the device that it is connected to.
Fig. 38: Typical Provisioning Process

1. Transport-specific discovery and connection
   - Some form of beaconsing
   - Client connects

2. Session Establishment
   - Get Version Request
   - Get Version Response
   - Session Setup Request
   - Session Setup Response
   - One or multiple steps as per protocol

3. Configuration
   - App-specific Set Config (optional)
   - Set Config Response (optional)
   - Wi-Fi SetConfig(SSID, Passphrase...)
   - Wi-Fi SetConfig response
   - Wi-Fi ApplyConfig cmd
   - Wi-Fi ApplyConfig resp
   - Wi-Fi GetStatus cmd (repeated)
   - Wi-Fi GetStatus resp (repeated)

4. Close connection
   - Close Connection
3. The device manufacturer may choose proof-of-possession (PoP), a unique per-device secret to be entered on the provisioning client as a security measure to make sure that only the user can provision the device in their possession.

There are two levels of security schemes, of which the developer may select one or a combination, depending on requirements.

1. **Transport Security**

For SoftAP provisioning, developers may choose WPA2-protected security with unique per-device passphrase. Unique per-device passphrase can also act as a proof-of-possession. For Bluetooth LE, the “just-works” security can be used as a transport-level security after assessing its provided level of security.

2. **Application Security**

The unified provisioning subsystem provides the application-level security (*Security 1 Scheme*) that provides data protection and authentication through PoP, 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 for advertisement 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 Bluetooth LE transport, device name or primary service included in the advertisement or a combination of both can be used for discovery.

**Architecture**

The below diagram shows the architecture of unified provisioning:

![Unified Provisioning Architecture Diagram](image-url)
Chapter 2. API Reference

It relies on the base layer called *Protocol Communication* (protocomm) which provides a framework for security schemes and transport mechanisms. The 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, the application can directly use protocomm to register custom handlers.

The 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 the logical channel for communication for specific type of information. For example, security handshake happens on a different endpoint from 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 the SoftAP+HTTP transport, the end-point corresponds to URI, whereas in case of Bluetooth LE, the end-point corresponds to the 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, the unified provisioning supports the following security schemes:

1. Security 0
   - No security (No encryption).
2. Security 1
   - 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.
3. Security 2
   - SRP6a-based shared key derivation and AES256-GCM mode encryption of the data.

*Note:* The respective security schemes need to be enabled through the project configuration menu. Please refer to *Enabling Protocomm Security Version* for more details.

**Security 1 Scheme**

The Security 1 scheme details are shown in the below sequence diagram:

**Security 2 Scheme**

The Security 2 scheme is based on the Secure Remote Password (SRP6a) protocol, see RFC 5054.

The protocol requires the Salt and Verifier to be generated beforehand with the help of the identifying username \( I \) and the plaintext password \( p \). The Salt and Verifier are then stored on ESP32-S2.

- The password \( p \) and the username \( I \) are to be provided to the Phone App (Provisioning entity) by suitable means, e.g., QR code sticker.

Details about the Security 2 scheme 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. Application implementation can be found as an example under *provisioning*. 
Fig. 40: Security 1

Client

Device

Generate Key Pair

\{cli\_privkey, cli\_pubkey\} = curve25519\_keygen()

SessionCmd0(cli\_pubkey)

Generate Key Pair

\{dev\_privkey, dev\_pubkey\} = curve25519\_keygen()

Initialization Vector

dev\_rand = gen\_16byte\_random()

Shared Key

\text{shared\_key}(\text{No PoP}) = \text{curve25519}(\text{dev\_privkey}, \text{cli\_pubkey})

\text{shared\_key}(\text{with PoP}) = \text{curve25519}(\text{dev\_privkey}, \text{cli\_pubkey}) \_\_\text{SHA256}(\text{pop})

SessionResp0(dev\_pubkey, dev\_rand)

Shared Key

\text{shared\_key}(\text{No PoP}) = \text{curve25519}(\text{cli\_privkey}, \text{dev\_pubkey})

\text{shared\_key}(\text{with PoP}) = \text{curve25519}(\text{cli\_privkey}, \text{dev\_pubkey}) \_\_\text{SHA256}(\text{pop})

Verification Token

cli\_verify = aes\_ctr\_enc(key=shared\_key, data=dev\_pubkey, nonce=dev\_rand)

SessionCmd1(cli\_verify)

Verify Client

check (dev\_pubkey == aes\_ctr\_dec(cli\_verify...))

Verify Device

dev\_verify = aes\_ctr\_enc(key=shared\_key, data=cli\_pubkey, nonce=(prev-context))

SessionResp1(dev\_verify)
Chapter 2. API Reference

Fig. 41: Security 2

Espressif Systems 1245 Release master

Submit Document Feedback
Provisioning Tools

Provisioning applications are available for various platforms, along with source code:

- **Android:**
  - Bluetooth LE Provisioning app on Play Store.
  - SoftAP Provisioning app on Play Store.
  - Source code on GitHub: esp-idf-provisioning-android.

- **iOS:**
  - Bluetooth LE Provisioning app on App Store.
  - SoftAP Provisioning app on App Store.
  - Source code on GitHub: 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 are thus more user centric, while the command-line application is useful as a debugging tool for developers.

2.7.3 Wi-Fi Provisioning

Overview

This component provides APIs that control the Wi-Fi provisioning service for receiving and configuring Wi-Fi credentials over SoftAP or Bluetooth LE transport via secure Protocol Communication sessions. The set of wifi_prov_mgr_ APIs help quickly implement a provisioning service that has 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 must be called prior to invoking any other wifi_prov_mgr_ APIs. Note that the manager relies on other components of ESP-IDF, namely NVS, TCP/IP, Event Loop and Wi-Fi, and optionally mDNS, hence these components must be initialized beforehand. The manager can be de-initialized at any moment by making a call to `wifi_prov_mgr_deinit()`.

```c
wifi_prov_mgr_config_t config = {
  .scheme = wifi_prov_scheme_ble,
  .scheme_event_handler = WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM
};
ESP_ERROR_CHECK( wifi_prov_mgr_init(config) );
```

The configuration structure `wifi_prov_mgr_config_t` has a few fields to specify the desired behavior of the manager:

- `wifi_prov_mgr_config_t.schema` - This is used to specify the provisioning scheme.
  Each scheme corresponds to one of the modes of transport supported by protocomm. Hence, support the following options:
  - `wifi_prov_scheme_ble` - Bluetooth LE transport and GATT Server for handling the provisioning commands.
  - `wifi_prov_scheme_softap` - Wi-Fi SoftAP transport and HTTP Server for handling the provisioning commands.
  - `wifi_prov_scheme_console` - Serial transport and console for handling the provisioning commands.
- `wifi_prov_mgr_config_t.scheme_event_handler` - An event handler defined along with the scheme. Choosing the appropriate scheme-specific event handler allows the manager to take care of certain matters automatically. Presently, this option is not used for either the SoftAP or Console-based provisioning, but is very convenient for Bluetooth LE. To understand how, we must recall that Bluetooth requires a substantial amount of memory to function, and once
the provisioning is finished, the main application may want to reclaim back this memory (or part of it) if it needs to use either Bluetooth LE or classic Bluetooth. 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 Bluetooth LE/classic Bluetooth/BTDM memory is freed automatically when the provisioning manager is de-initialized. The available options are:

- `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM` - Free both classic Bluetooth and Bluetooth LE/BTDM memory. Used when the main application does not require Bluetooth at all.
- `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE` - Free only Bluetooth LE memory. Used when main application requires classic Bluetooth.
- `WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT` - Free only classic Bluetooth. Used when main application requires Bluetooth LE. In this case freeing happens right when the manager is initialized.
- `WIFI_PROV_EVENT_HANDLER_NONE` - Do not use any scheme specific handler. Used when the provisioning scheme is not Bluetooth LE, i.e., using SoftAP or Console, or when main application wants to handle the memory reclaiming on its own, or needs both Bluetooth LE and classic Bluetooth to function.

- `wifi_prov_mgr_config_t::app_event_handler` (Deprecated) - It is now recommended to catch `WIFI_PROV_EVENT` that is 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:

```c
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_t *)event_data;
            ESP_LOGE(TAG, "Provisioning failed!\n\tReason : %s"
            "\n\tPlease reset to factory and retry\n\tprovisioning",
            (*reason == WIFI_PROV_STA_AUTH_ERROR) ? "Wi-Fi station authentication failed" : "Wi-Fi\n\taccess-point not found");
            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;
        }
    }
```

(continues on next page)
The manager can be de-initialized at any moment by making a call to `wifi_prov_mgr_deinit()`.  

**Check the Provisioning State**  
Whether the 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 the 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 the 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
- The main application must implement some logic to call `esp_wifi_` APIs for erasing the credentials at runtime
- The main application must implement some logic to force start the provisioning irrespective of the provisioning state

```c
bool provisioned = false;
ESP_ERROR_CHECK( wifi_prov_mgr_is_provisioned(&provisioned) );
```

**Start the Provisioning Service**  
At the time of starting provisioning we need to specify a service name and the corresponding key, that is to say:

- A Wi-Fi SoftAP SSID and a passphrase, respectively, when the scheme is `wifi_prov_scheme_softap`.
- Bluetooth LE device name with the service key ignored when the scheme is `wifi_prov_scheme_ble`.

Also, since internally the manager uses `procomm`, 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 or decryption of subsequent messages.
- Security 0 is simply plain text communication. In this case the `pop` is simply ignored.

See *Unified Provisioning* for details about the security features.

```c
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_ERROR_CHECK( wifi_prov_mgr_start_provisioning(security, pop, service_name, service_key) );
```

The provisioning service automatically finishes only if it receives valid Wi-Fi AP credentials followed by successful connection of device to the AP with 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 does not accept new credentials anymore, but the provisioning service keeps on running, only to convey failure to the client, until the device is restarted. Upon restart, the provisioning state turns out to be true this time, as credentials are found in NVS, but the device does fail again 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 the Provisioning State*.
Waiting for Completion  Typically, the main application waits for the provisioning to finish, then de-initializes the manager to free up resources, and finally starts 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_ERROR_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` and call `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 the manager once the 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 Bluetooth LE 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 host name set by the main application is used, and the service type is internally set to `_esp_wifi_prov`.

When using Bluetooth LE transport, a custom 128-bit UUID should be set using `wifi_prov_scheme_ble_set_service_uuid()`. This UUID is to be included in the Bluetooth LE advertisement and corresponds to the primary GATT service that provides provisioning endpoints as GATT characteristics. Each GATT characteristic is formed using the primary service UUID as the base, with different auto-assigned 12th and 13th bytes, presumably counting from the 0th byte. Since an endpoint characteristic UUID is auto-assigned, it should not 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 is 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 9: Endpoints Provided by the Provisioning Service

<table>
<thead>
<tr>
<th>Endpoint Name i.e., Bluetooth LE + GATT Server</th>
<th>URI, i.e., SoftAP + HTTP Server + mDNS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prov-session</td>
<td>http://&lt;mdns-hostname&gt;.local/prov-session</td>
<td>Security endpoint used for session establishment</td>
</tr>
<tr>
<td>prov-scan</td>
<td><a href="http://wifi-prov.local/prov-scan">http://wifi-prov.local/prov-scan</a></td>
<td>the endpoint used for starting Wi-Fi scan and receiving scan results</td>
</tr>
<tr>
<td>prov-ctrl</td>
<td><a href="http://wifi-prov.local/prov-ctrl">http://wifi-prov.local/prov-ctrl</a></td>
<td>the endpoint used for controlling Wi-Fi provisioning state</td>
</tr>
<tr>
<td>prov-config</td>
<td>http://&lt;mdns-hostname&gt;.local/prov-config</td>
<td>the endpoint used for configuring Wi-Fi credentials on device</td>
</tr>
<tr>
<td>proto-ver</td>
<td>http://&lt;mdns-hostname&gt;.local/proto-ver</td>
<td>the endpoint for retrieving version info</td>
</tr>
</tbody>
</table>

Immediately after connecting, the client application may fetch the version/capabilities information from the `proto-ver` endpoint. All communications to this endpoint are unencrypted, hence necessary information, which 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 the `no_pop` capability is supported, which indicates that the service does not require proof of possession for authentication. Any application-related version or capabilities are given by other labels, e.g., `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, which is not needed when the 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 with the corresponding `.proto` files that can be found under `wifi_provisioning/proto`:

- **get_status**: For querying the Wi-Fi connection status. The device responds with a status which is one of connecting, connected or disconnected. If the status is disconnected, a disconnection reason is also to 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 starting the Wi-Fi station.

After session establishment, the 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, the scan is started in passive mode, which 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 120 ms 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 waits for at least 120 ms after completing the scan on a group of channels, and thus allows 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 3 creates 5 groups, with each group having 3 channels, except the last
one which has 14 % 3 = 2 channels. So, when the scan is started, the first 3 channels will be scanned, followed by a 120 ms 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 a few channels in a group may increase 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., Bluetooth LE, this can be safely set to 0, and hence achieve the shortest overall scanning time.

- **period_ms** (input) - The scan parameter specifying how long to wait on each channel.

- **scan_status** - It gives the status of scanning process:
  - **scan_finished** (output) - When the scan has finished, this returns true.
  - **result_count** (output) - This gives the total number of results obtained till now. If the scan is yet happening, this number keeps on updating.

- **scan_result** - For fetching the scan results. This can be called even if the scan is still on going.
  - **start_index** (input) - Where the index starts from to fetch the entries from the results list.
  - **count** (input) - The number of entries to fetch from the starting index.
  - **entries** (output) - The list of entries returned. Each entry consists of ssid, channel and rssi information.

The client can also control the provisioning state of the device using **wifi_ctrl** endpoint. The **wifi_ctrl** endpoint supports the following protobuf commands:

- **ctrl_reset** - Resets internal state machine of the device and clears provisioned credentials only in case of provisioning failures.
- **ctrl_reprov** - Resets internal state machine of the device and clears provisioned credentials only in case the device is to be provisioned again for new credentials after a previous successful provisioning.

**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 Protocol Communication 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.

```c
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 the 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 stops, and Bluetooth LE or SoftAP turns off, automatically after responding to the next get_status command. If get_status command is not received by the device, the service stops after a 30 s timeout.

On the other hand, if device is not able to connect using the provided Wi-Fi credentials, due to incorrect SSID or passphrase, the service keeps running, and get_status keeps responding with disconnected status and reason for disconnection. Any further attempts to provide another set of Wi-Fi credentials, are to be rejected. These credentials are 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 stops only 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 de-
vice to connect to some cloud service and obtain another set of credentials, and exchange these credentials
over a custom protocomm endpoint, then after successfully doing so, stop the provisioning service by calling
\texttt{wifi_prov_mgr_stop_provisioning()} inside the protocomm handler itself. The right amount of de-
lay ensures that the transport resources are freed only after the response from the protocomm handler reaches the
client side application.

\section*{Application Examples}

For complete example implementation see \texttt{provisioning/wifi_prov_mgr}.

\section*{Provisioning Tools}

Provisioning applications are available for various platforms, along with source code:

\begin{itemize}
\item **Android:**
  \begin{itemize}
  \item Bluetooth LE Provisioning app on Play Store.
  \item SoftAP Provisioning app on Play Store.
  \item Source code on GitHub: \texttt{esp-idf-provisioning-android}.
  \end{itemize}
\item **iOS:**
  \begin{itemize}
  \item Bluetooth LE Provisioning app on App Store.
  \item SoftAP Provisioning app on App Store.
  \item Source code on GitHub: \texttt{esp-idf-provisioning-ios}.
  \end{itemize}
\item **Linux/MacOS/Windows:** \texttt{tools/esp_prov}, a Python-based command-line tool for provisioning.
\end{itemize}

The phone applications offer simple UI and are thus more user centric, while the command-line application is useful
as a debugging tool for developers.

\section*{API Reference}

\subsection*{Header File}

\begin{itemize}
\item \texttt{components/wifi_provisioning/include/wifi_provisioning/manager.h}
\item This header file can be included with:

\begin{verbatim}
#include "wifi_provisioning/manager.h"
\end{verbatim}
\item This header file is a part of the API provided by the \texttt{wifi_provisioning} component. To declare that
your component depends on \texttt{wifi_provisioning}, add the following to your \texttt{CMakeLists.txt}:

\begin{verbatim}
REQUIRES wifi_provisioning
\end{verbatim}

or

\begin{verbatim}
PRIV_REQUIRES wifi_provisioning
\end{verbatim}
\end{itemize}

\subsection*{Functions}

\begin{verbatim}
esp_err_t wifi_prov_mgr_init (wifi_prov_mgr_config_t config)
\end{verbatim}

Initialize provisioning manager instance.

Configures the manager and allocates internal resources

Configuration specifies the provisioning scheme (transport) and event handlers

Event \texttt{WIFI_PROV_INIT} is emitted right after initialization is complete

\begin{verbatim}
Parameters config --[in] Configuration structure
\end{verbatim}

\begin{verbatim}
Returns
\item ESP_OK : Success
\end{verbatim}
Chapter 2. API Reference

• ESP_FAIL : Fail

**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.

---

**Parameters**

- `provisioned` – [out] True if provisioned, else false

**Returns**

- ESP_OK : Retrieved provision state successfully
- ESP_FAIL : Wi-Fi not initialized
- ESP_ERR_INVALID_ARG : Null argument supplied

**esp_err_t wifi_prov_mgr_start_provisioning (wifi_prov_security_t security, const void *wifi_prov_sec_params, 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

---

**Parameters**

- `security` – [in] 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
  - WIFI_PROV_SECURITY_2 : SRP6a based authentication and key exchange followed by AES-GCM encryption/decryption of provisioning messages
- `wifi_prov_sec_params` – [in] Pointer to security params (NULL if not needed). This is not needed for protocomm security 0. This pointer should hold the struct of type wifi_prov_security1_params_t for protocomm security 1 and wifi_prov_security2_params_t for protocomm security 2 respectively. This pointer and
its contents should be valid till the provisioning service is running and has not been stopped or de-inited.

• **service_name** *[in]* Unique name of the service. This translates to:
  – Wi-Fi SSID when provisioning mode is softAP
  – Device name when provisioning mode is BLE

• **service_key** *[in]* 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

**Returns**

- ESP_OK: Provisioning started successfully
- ESP_FAIL: Failed to start provisioning service
- ESP_ERR_INVALID_STATE: Provisioning manager not initialized or already started

```c
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 protocol 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 protocol service is stopped.

Cleanup_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.

```c
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.

```c
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 protocol 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 up to 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()
Chapter 2. API Reference

- ESP_OK : Success
- ESP_ERR_INVALID_STATE : Manager not initialized or provisioning service already started

`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()

**Parameters**
- `label` [in] String indicating the application name.
- `version` [in] String indicating the application version. There is no constraint on format.
- `capabilities` [in] Array of strings with capabilities. These could be used by the client side app to know the application registered endpoint capabilities
- `total_capabilities` [in] Size of capabilities array

**Returns**
- 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

`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

**Parameters**
- `ep_name` [in] unique name of the endpoint

**Returns**
- ESP_OK : Success
- ESP_FAIL: Failure
**esp_err_t wifi_prov_mgr_endpoint_register** (const char* `ep_name`, `protocomm_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

**Parameters**
- `ep_name` [in] Name of the endpoint
- `handler` [in] Endpoint handler function
- `user_ctx` [in] User data

**Returns**
- ESP_OK : Success
- ESP_FAIL : Failure

**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** `ep_name` [in] Name of the endpoint

**esp_err_t wifi_prov_mgr_get_wifi_state** (`wifi_prov_sta_state_t`*state*)

Get state of Wi-Fi Station during provisioning.

**Parameters** `state` [out] Pointer to `wifi_prov_sta_state_t` variable to be filled

**Returns**
- ESP_OK : Successfully retrieved Wi-Fi state
- ESP_FAIL : Provisioning app not running

**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.

**Parameters** `reason` [out] Pointer to `wifi_prov_sta_fail_reason_t` variable to be filled

**Returns**
- ESP_OK : Successfully retrieved Wi-Fi disconnect reason
- ESP_FAIL : Provisioning app not running

**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

**Parameters wifi_cfg – [in]** Pointer to Wi-Fi configuration structure

**Returns**
- ESP_OK : Wi-Fi configured and started successfully
- ESP_FAIL : Failed to set configuration

**esp_err_t wifi_prov_mgr_reset_provisioning** (void)

Reset Wi-Fi provisioning config.

Calling this API will restore WiFi stack persistent settings to default values.

**Returns**
- ESP_OK : Reset provisioning config successfully
- ESP_FAIL : Failed to reset provisioning config

**esp_err_t wifi_prov_mgr_reset_sm_state_on_failure** (void)

Reset internal state machine and clear provisioned credentials.

This API should be used to restart provisioning ONLY in the case of provisioning failures without rebooting the device.

**Returns**
- ESP_OK : Reset provisioning state machine successfully
- ESP_FAIL: Failed to reset provisioning state machine
- ESP_ERR_INVALID_STATE : Manager not initialized

**esp_err_t** wifi_prov_mgr_reset_sm_state_for_reprovision (void)

Reset internal state machine and clear provisioned credentials.

This API can be used to restart provisioning ONLY in case the device is to be provisioned again for new credentials after a previous successful provisioning without rebooting the device.

**Note:** This API can be used only if provisioning auto-stop has been disabled using wifi_prov_mgr_disable_auto_stop()

**Returns**
- ESP_OK : Reset provisioning state machine successfully
- ESP_FAIL : Failed to reset provisioning state machine
- ESP_ERR_INVALID_STATE : Manager not initialized

**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_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

typedef protocomm_security2_params_t wifi_prov_security2_params_t
Security 2 params structure This needs to be passed when using WIFI_PROV_SECURITY_2.

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:
enumerator **WIFI_PROV_INIT**

Emitted when the manager is initialized

denumerator **WIFI_PROV_START**

Indicates that provisioning has started

denumerator **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

denumerator **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

denumerator **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

denumerator **WIFI_PROV_END**

Signals that provisioning service has stopped

denumerator **WIFI_PROV_DEINIT**

Signals that manager has been de-initialized

denum **wifi_prov_security**

Security modes supported by the Provisioning Manager.

These are same as the security modes provided by protocomm

*Values:*

enumerator **WIFI_PROV_SECURITY_0**

No security (plain-text communication)

denumerator **WIFI_PROV_SECURITY_1**

This secure communication mode consists of X25519 key exchange

- proof of possession (pop) based authentication
- AES-CTR encryption

denumerator **WIFI_PROV_SECURITY_2**

This secure communication mode consists of SRP6a based authentication and key exchange

- AES-GCM encryption/decryption

**Header File**

- components/wifi_provisioning/include/wifi_provisioning/scheme_ble.h

This header file can be included with:

```c
#include "wifi_provisioning/scheme_ble.h"
```

- This header file is a part of the API provided by the wifi_provisioning component. To declare that your component depends on wifi_provisioning, add the following to your CMakeLists.txt:
Functions

void wifi_prov_scheme_ble_event_cb_free_btdm (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 0000fff-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 at least until provisioning is started. Upon start, the manager will store an internal copy of this UUID, and this data can be freed or invalidated afterwards.

Parameters

- uuid128 - [in] A custom 128 bit UUID

Returns

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null argument

esp_err_t wifi_prov_scheme_ble_set_mfg_data (uint8_t *mfg_data, ssize_t mfg_data_len)

Set manufacturer specific data in scan response.

This must be called before starting provisioning, i.e. before making a call to wifi_prov_mgr_start_provisioning().

Note: It is important to understand that length of custom manufacturer data should be within limits. The manufacturer data goes into scan response along with BLE device name. By default, BLE device name length is of 11 bytes, however it can vary as per application use case. So, one has to honour the scan response data size limits i.e. (mfg_data_len + 2) < 31 - (device_name_length + 2). If the mfg data length exceeds this limit, the length will be truncated.

Parameters

- mfg_data - [in] Custom manufacturer data
- mfg_data_len - [in] Manufacturer data length

Returns

- ESP_OK : Success
- ESP_ERR_INVALID_ARG : Null argument

Macros

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM
Chapter 2. API Reference

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE

WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT

Header File
- components/wifi_provisioning/include/wifi_provisioning/scheme_softap.h
- This header file can be included with:
  ```
  #include "wifi_provisioning/scheme_softap.h"
  ```
- This header file is a part of the API provided by the wifi_provisioning component. To declare that your component depends on wifi_provisioning, add the following to your CMakeLists.txt:
  ```
  REQUIRE wifi_provisioning
  ```
  or
  ```
  PRIV_REQUIRES wifi_provisioning
  ```

Functions

```
void wifi_prov_scheme_softap_set_httpd_handle (void *handle)
```
Provide HTTPD Server handle externally.
Useful in cases wherein applications need the webservice 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 handle -[in] Handle to HTTPD server instance

Header File
- components/wifi_provisioning/include/wifi_provisioning/scheme_console.h
- This header file can be included with:
  ```
  #include "wifi_provisioning/scheme_console.h"
  ```
- This header file is a part of the API provided by the wifi_provisioning component. To declare that your component depends on wifi_provisioning, add the following to your CMakeLists.txt:
  ```
  REQUIRE wifi_provisioning
  ```
  or
  ```
  PRIV_REQUIRES wifi_provisioning
  ```

Header File
- components/wifi_provisioning/include/wifi_provisioning/wifi_config.h
- This header file can be included with:
  ```
  #include "wifi_provisioning/wifi_config.h"
  ```
- This header file is a part of the API provided by the wifi_provisioning component. To declare that your component depends on wifi_provisioning, add the following to your CMakeLists.txt:
**Functions**

```c
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**

```c
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 AP to which connection was established
- `uint8_t channel`
  - Channel of the AP
- `uint8_t auth_mode`
  - Authorization mode of the AP

```c
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 protocol requests from master.
This is to be passed as priv_data for protocol 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:
enumerator WIFI_PROV_STA_CONNECTING
enumerator WIFI_PROV_STA_CONNECTED
enumerator WIFI_PROV_STA_DISCONNECTED

enum wifi_prov_sta_fail_reason_t
WiFi STA connection fail reason.
Values:
enumerator WIFI_PROV_STA_AUTH_ERROR
enumerator WIFI_PROV_STA_AP_NOT_FOUND

Code examples for above API are provided in the provisioning directory of ESP-IDF examples.
Code example for above API is provided in wifi/smart_config.
Code example for above API is provided in wifi/wifi_easy_connect/dpp-enrollee.

2.8 Storage API

This section contains reference of the high-level storage APIs. They are based on low-level drivers such as SPI flash, SD/MMC.

- Partitions API allow block based access to SPI flash according to the Partition Tables.
- Non-Volatile Storage library (NVS) implements a fault-tolerant wear-levelled key-value storage in SPI NOR flash.
- Virtual File System (VFS) library provides an interface for registration of file system drivers. SPIFFS, FAT and various other file system libraries are based on the VFS.
- SPIFFS is a wear-levelled file system optimized for SPI NOR flash, well suited for small partition sizes and low throughput
- FAT is a standard file system which can be used in SPI flash or on SD/MMC cards
- Wear Levelling library implements a flash translation layer (FTL) suitable for SPI NOR flash. It is used as a container for FAT partitions in flash.

Note: It is suggested to use high-level APIs (esp_partition or file system) instead of low-level driver APIs to access the SPI NOR flash.
Due to the restriction of NOR flash and ESP hardware, accessing the main flash will affect the performance of the whole system. See SPI Flash API to learn more about the limitations.

2.8.1 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 receives 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.
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"). The filesystem uses 8.3 filenames format (SFN) by default. If you need to use long filenames (LFN), enable the CONFIG_FATFS_LONG_FILENAMES option. More details on the FatFs filenames are available here.
5. Optionally, by enabling the option CONFIG_FATFS_USE_FASTSEEK, you can enable the POSIX lseek function to perform it faster. The fast seek does not work for files in write mode, so to take advantage of fast seek, you should open (or close and then reopen) the file in read-only mode.
6. Optionally, by enabling the option CONFIG_FATFS_IMMEDIATE_FSYNC, you can enable automatic calling of f_sync() to flush recent file changes after each call of vfs_fat_write(), vfs_fat_pwrite(), vfs_fat_link(), vfs_fat_truncate() and vfs_fat_ftruncate() functions. This feature improves file-consistency and size reporting accuracy for the FatFs, at a price on decreased performance due to frequent disk operations.
7. Optionally, call the FatFs library functions directly. In this case, use paths without a VFS prefix, for example, "/hello.txt".
8. Close all open files.
9. Call the FatFs function f_mount() for the same drive number with NULL FATFS* argument to unmount the filesystem.
10. 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.
11. 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()`, `esp_vfs_fat_sdspi_mount()`, and `esp_vfs_fat_sdcardUnmount()` wrap the steps described above and also handle SD card initialization. These functions are described in the next section.

**Using FatFs with VFS and SD Cards**

The header file `fatfs/vfs/esp_vfs_fat.h` defines convenience functions `esp_vfs_fat_sdmmc_mount()`, `esp_vfs_fat_sdspi_mount()`, and `esp_vfs_fat_sdcardUnmount()`. These functions 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_sdmmcUnmount()` unmounts the filesystem and releases the resources acquired by `esp_vfs_fat_sdmmc_mount()`.

**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_spiflash_mount_ro()` and `esp_vfs_fat_spiflashUnmount_ro()`. 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.

**FatFS Disk IO Layer**

FatFs has been extended with API functions that register the disk I/O driver at runtime. These APIs 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()`.

```c
void ff_diskio_register (BYTE pdrv, const ff_diskio_impl_t *discio_impl)
```

Register or unregister diskio driver for given drive number.

*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

```c
struct ff_diskio_impl_t
```

Structure of pointers to disk IO driver functions.

See FatFs documentation for details about these functions

**Public Members**

```c
DSTATUS (*init)(unsigned char pdrv)
```

disk initialization function

```c
DSTATUS (*status)(unsigned char pdrv)
```

disk status check function

```c
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.

FatFs Partition Generator

We provide a partition generator for FatFs (wl_fatfsgen.py) which is integrated into the build system and could be easily used in the user project.

The tool is used to create filesystem images on a host and populate it with content of the specified host folder.

The script is based on the partition generator (fatfsgen.py). Apart from generating partition, it can also initialize wear levelling.

The latest version supports both short and long file names, FAT12 and FAT16. The long file names are limited to 255 characters and can contain multiple periods (.) characters within the filename and additional characters +, , ; , =, [ and ].

An in-depth description of the FatFs partition generator and analyzer can be found at Generating and parsing FAT partition on host.

Build System Integration with FatFs Partition Generator

It is possible to invoke FatFs generator directly from the CMake build system by calling fatfs_create_spiflash_image:

```
fatfs_create_spiflash_image(<partition> <base_dir> [FLASH_IN_PROJECT])
```

If you prefer generating partition without wear levelling support, you can use fatfs_create_rawflash_image:

```
fatfs_create_rawflash_image(<partition> <base_dir> [FLASH_IN_PROJECT])
```

fatfs_create_spiflash_image respectively fatfs_create_rawflash_image must be called from project’s CMakeLists.txt.

If you decide for any reason to use fatfs_create_rawflash_image (without wear levelling support), beware that it supports mounting only in read-only mode in the device.
The arguments of the function are as follows:

1. partition - the name of the partition as defined in the partition table (e.g., storage/fatfsgen/partitions_example.csv).
2. base_dir - the directory that will be encoded to FatFs partition and optionally flashed into the device. Beware that you have to specify the suitable size of the partition in the partition table.
3. flag FLASH_IN_PROJECT - optionally, users can have the image automatically flashed together with the app binaries, partition tables, etc. on idf.py flash -p <PORT> by specifying FLASH_IN_PROJECT.
4. flag PRESERVE_TIME - optionally, users can force preserving the timestamps from the source folder to the target image. Without preserving the time, every timestamp will be set to the FATFS default initial time (1st January 1980).

For example:

```
fatfs_create_spiflash_image(my_fatfs_partition my_folder FLASH_IN_PROJECT)
```

If FLASH_IN_PROJECT is not specified, the image will still be generated, but you will have to flash it manually using esptool.py or a custom build system target.

For an example, see storage/fatfsgen.

### FatFs Partition Analyzer

(fatfsparse.py) is a partition analyzing tool for FatFs.

It is a reverse tool of (fatfsgen.py), i.e., it can generate the folder structure on the host based on the FatFs image.

Usage:

```
./fatfsparse.py [-h] [--wl-layer {detect,enabled,disabled}] fatfs_image.img
```

### High-level API Reference

#### Header File

- components/fatfs/vfs/esp_vfs_fat.h
- This header file can be included with:

  ```
  #include "esp_vfs_fat.h"
  ```

- This header file is a part of the API provided by the fatfs component. To declare that your component depends on fatfs, add the following to your CMakeLists.txt:

  ```
  REQUIRES fatfs
  ```
  or

  ```
  PRIV_REQUIRES fatfs
  ```

#### Functions

**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.

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_fs` [out] pointer to FATFS structure which can be used for FATFS f_mount call is returned via this argument.

Returns
- 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

```c
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.

Parameters `base_path` – path prefix where FATFS is registered. This is the same used when esp_vfs_fat_register was called.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE if FATFS is not registered in VFS

```c
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.

Note: Use this API to mount a card through SDSPI is deprecated. Please call esp_vfs_fat_sdspi_mount() instead for that case.

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 SD_SPI_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.
**mount_config** - pointer to structure with extra parameters for mounting FATFS

**out_card** [out] if not NULL, pointer to the card information structure will be returned via this argument

**Returns**

- 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

```c
esp_err_t esp_vfs_fat_sdspi_mount
    (const char*base_path,const sdmmc_host_t*host_config_input,
     const sdspi_device_config_t*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 an SPI Master device based on the SPI Master driver with configuration in slot_config, and attach it to an initialized SPI bus.
- initializes SD card with configuration in host_config_input
- 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.

**Note:** This function try to attach the new SD SPI device to the bus specified in host_config. Make sure the SPI bus specified in host_config->slot have been initialized by `spi_bus_initialize()` before.

**Parameters**

- **base_path** - path where partition should be registered (e.g. “/sdcard”)
- **host_config_input** - Pointer to structure describing SDMMC host. This structure can be initialized using SDMMC_HOST_DEFAULT() macro.
- **slot_config** - Pointer to structure with slot configuration. For SPI peripheral, pass a pointer to sdspi_device_config_t structure initialized using SD- SPI_DEVICE_CONFIG_DEFAULT().
- **mount_config** - pointer to structure with extra parameters for mounting FATFS
- **out_card** [out] If not NULL, pointer to the card information structure will be returned via this argument. It is suggested to hold this handle and use it to unmount the card later if needed. Otherwise it’s not suggested to use more than one card at the same time and unmount one of them in your application.

**Returns**

- 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

```c
esp_err_t esp_vfs_fat_sdmmc_unmount
    (void)
```

Unmount FAT filesystem and release resources acquired using esp_vfs_fat_sdmmc_mount.

**Deprecated:**

Use `esp_vfs_fat_sdcard_unmount()` instead.
Chapter 2. API Reference

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_sdmmc_mount hasn’t been called

`esp_err_t esp_vfs_fat_sdmmc_unmount` (const char *base_path, sdmmc_card_t *card)

Unmount an SD card from the FAT filesystem and release resources acquired using `esp_vfs_fat_sdmmc_mount()` or `esp_vfs_fat_sdspi_mount()`

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the card argument is unregistered
- ESP_ERR_INVALID_STATE if esp_vfs_fat_sdmmc_mount hasn’t been called

`esp_err_t esp_vfs_fat_sdcard_format` (const char *base_path, sdmmc_card_t *card)

Format FAT filesystem.

Note: This API should be only called when the FAT is already mounted.

Parameters

- `base_path` – Path where partition should be registered (e.g. “/sdcard”)
- `card` – Pointer to the card handle, which should be initialised by calling `esp_vfs_fat_sdspi_mount` first

Returns

- ESP_OK
- ESP_ERR_INVALID_STATE: FAT partition isn’t mounted, call `esp_vfs_fat_sdmmc_mount` or `esp_vfs_fat_sdspi_mount` first
- ESP_ERR_NO_MEM: if memory cannot be allocated
- ESP_FAIL: fail to format it, or fail to mount back

`esp_err_t esp_vfs_fat_spiflash_mount_rw_wl` (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.

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
- `wl_handle` – [out] wear levelling driver handle

Returns

- 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_rw_wl 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
\texttt{esp_err_t esp_vfs_fat_spiflash_unmount_rw_wl} (const char \*base_path, \texttt{wl_handle} \*wl_handle)

Unmount FAT filesystem and release resources acquired using \texttt{esp_vfs_fat_spiflash_mount_rw_wl}.

**Parameters**
- \texttt{base_path} – path where partition should be registered (e.g. “/spiflash”)
- \texttt{wl_handle} – wear levelling driver handle returned by \texttt{esp_vfs_fat_spiflash_mount_rw_wl}

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_STATE if \texttt{esp_vfs_fat_spiflash_mount_rw_wl} hasn’t been called

\texttt{esp_err_t esp_vfs_fat_spiflash_format_rw_wl} (const char \*base_path, const char \*partition_label)

Format FAT filesystem.

**Note:** This API can be called when the FAT is mounted / not mounted. If this API is called when the FAT isn’t mounted (by calling \texttt{esp_vfs_fat_spiflash_mount_rw_wl}), this API will first mount the FAT then format it, then restore back to the original state.

**Parameters**
- \texttt{base_path} – Path where partition should be registered (e.g. “/spiflash”)
- \texttt{partition_label} – Label of the partition which should be used

**Returns**
- ESP_OK
- ESP_ERR_NO_MEM: if memory cannot be allocated
- Other errors from \texttt{esp_vfs_fat_spiflash_mount_rw_wl}

\texttt{esp_err_t esp_vfs_fat_spiflash_mount_ro} (const char \*base_path, const char \*partition_label, const \texttt{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.

**Parameters**
- \texttt{base_path} – path where FATFS partition should be mounted (e.g. “/spiflash”)
- \texttt{partition_label} – label of the partition which should be used
- \texttt{mount_config} – pointer to structure with extra parameters for mounting FATFS

**Returns**
- 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 \texttt{esp_vfs_fat_spiflash_mount_ro} 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

\texttt{esp_err_t esp_vfs_fat_spiflash_unmount_ro} (const char \*base_path, const char \*partition_label)

Unmount FAT filesystem and release resources acquired using \texttt{esp_vfs_fat_spiflash_mount_ro}.

---

Espressif Systems

Submit Document Feedback
Parameters

- base_path - path where partition should be registered (e.g. “/spiflash”)
- partition_label - label of partition to be unmounted

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if esp_vfs_fat_spiflash_mount_ro hasn’t been called

```
esp_err_t esp_vfs_fat_info (const char*base_path, uint64_t*out_total_bytes, uint64_t*out_free_bytes)
```

Get information for FATFS partition.

Parameters

- base_path - Base path of the partition examined (e.g. “/spiflash”)
- out_total_bytes - [out] Size of the file system
- out_free_bytes - [out] Free bytes available in the file system

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE if partition not found
- ESP_FAIL if another FRESULT error (saved in errno)

Structures

```
struct esp_vfs_fat_mount_config_t
```

Configuration arguments for esp_vfs_fat_sdmmc_mount and esp_vfs_fat_spiflash_mount_rw_wl 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 * sector size. For SD cards, sector size is always 512 bytes. For wearlevelling, 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.

```
bool disk_status_check_enable
```

Enables real ff_disk_status function implementation for SD cards (ff_sdmmc_status). Possibly slows down IO performance.

Try to enable if you need to handle situations when SD cards are not unmounted properly before physical removal or you are experiencing issues with SD cards.

Doesn’t do anything for other memory storage media.

Type Definitions

```
typedef esp_vfs_fat_mount_config_t esp_vfs_fat_sdmmc_mount_config_t
```
### 2.8.2 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)*
  - Download and unzip it, and follow the instructions inside the `doc` folder.
- Direct flash programming using custom production tools.

**Prerequisites**

This utility is dependent on ESP-IDF’s NVS Partition Generator Utility.

- **Operating System requirements:**
  - Linux / MacOS / Windows (standard distributions)
- **The following packages are needed to use this utility:**
  - Python

**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):

```plaintext
name1,namespace,  <!-- First entry should be of type "namespace"
key1,type1,encoding1
key2,type2,encoding2,REPEAT
name2,namespace,
key3,type3,encoding3
key4,type4,encoding4
```
Chapter 2. API Reference

**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
2,1a2b3c4d5e6fcddd,,102
3,1a2b3c4d5e6ffeeff,,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:
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:**

<table>
<thead>
<tr>
<th>No.</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-h/--help</td>
<td>Show the help message and exit</td>
</tr>
</tbody>
</table>

**Commands:**

Run mfg_gen.py {command} -h for additional help

<table>
<thead>
<tr>
<th>No.</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>generate</td>
<td>Generate NVS partition</td>
</tr>
<tr>
<td>2</td>
<td>generate-key</td>
<td>Generate keys for encryption</td>
</tr>
</tbody>
</table>

**To generate factory images for each device (Default):**

**Usage:**

```
```

**Positional Arguments:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf</td>
<td>Path to configuration csv file to parse</td>
</tr>
<tr>
<td>values</td>
<td>Path to values csv file to parse</td>
</tr>
<tr>
<td>prefix</td>
<td>Unique name for each output filename prefix</td>
</tr>
<tr>
<td>size</td>
<td>Size of NVS partition in bytes (must be multiple of 4096)</td>
</tr>
</tbody>
</table>

**Optional Arguments:**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h/--help</td>
<td>Show the help message and exit</td>
</tr>
<tr>
<td>--fileid FILEID</td>
<td>Unique file identifier (any key in values file) for each filename suffix</td>
</tr>
<tr>
<td></td>
<td>(Default: numeric value(1,2,3…))</td>
</tr>
<tr>
<td>--version (1,2)</td>
<td>Set multipage blob version. (Default: Version 2)</td>
</tr>
<tr>
<td></td>
<td>Version 1 - Multipage blob support disabled.</td>
</tr>
<tr>
<td></td>
<td>Version 2 - Multipage blob support enabled</td>
</tr>
<tr>
<td>--keygen</td>
<td>Generates key for encrypting NVS partition</td>
</tr>
<tr>
<td>--inputkey INPUTKEY</td>
<td>File having key for encrypting NVS partition</td>
</tr>
<tr>
<td>--outdir OUTDIR</td>
<td>Output directory to store files created (Default: current directory)</td>
</tr>
<tr>
<td>--key_protect_hmac</td>
<td>If set, the NVS encryption key protection scheme based on HMAC peripheral</td>
</tr>
<tr>
<td></td>
<td>is used; else the default scheme based on Flash Encryption is used</td>
</tr>
<tr>
<td>--kp_hmac_keygen</td>
<td>Generate the HMAC key for HMAC-based encryption scheme</td>
</tr>
<tr>
<td>--kp_hmac_keyfile</td>
<td>Path to output HMAC key file</td>
</tr>
<tr>
<td>--kp_hmac_inputkey</td>
<td>File having the HMAC key for generating the NVS encryption keys</td>
</tr>
</tbody>
</table>

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:

```bash
python mfg_gen.py generate samples/sample_config.csv samples/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:
  ```bash
  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. 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.

- To generate an encrypted image using the HMAC-based scheme, the above command can be used along with some additional parameters.
  - Encrypt by allowing the utility to generate encryption keys and the HMAC-key:
    ```bash
    python mfg_gen.py generate samples/sample_config.csv samples/sample_values_singlepage_blob.csv Sample 0x3000 --keygen --key--protect_hmac --kp_hmac_keygen
    ```

**Note:** Encryption key of the format `<outdir>/keys/keys--<timestamp>.bin` and HMAC key of the format `<outdir>/keys/hmac-keys--<timestamp>.bin` are created.

- Encrypt by allowing the utility to generate encryption keys with user-provided HMAC-key:
  ```bash
  python mfg_gen.py generate samples/sample_config.csv samples/sample_values_singlepage_blob.csv Sample 0x3000 --keygen --key--protect_hmac --kp_hmac--inputkey testdata/sample_hmac_key.bin
  ```

(continues on next page)
Note: You can provide the custom filename for the HMAC key as well as the encryption key as a parameter.

- Encrypt by providing the encryption keys as input binary file:

```bash
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code>/--help</td>
<td>Show the help message and exit</td>
</tr>
<tr>
<td>--keyfile KEYFILE</td>
<td>Path to output encryption keys file</td>
</tr>
<tr>
<td>--outdir OUTDIR</td>
<td>Output directory to store files created. (Default: current directory)</td>
</tr>
<tr>
<td>--key_protect_hmac</td>
<td>If set, the NVS encryption key protection scheme based on HMAC peripheral is used; else the default scheme based on Flash Encryption is used</td>
</tr>
<tr>
<td>--kp_hmac_keygen</td>
<td>Generate the HMAC key for HMAC-based encryption scheme</td>
</tr>
<tr>
<td>--kp_hmac_keyfile</td>
<td>Path to output HMAC key file</td>
</tr>
<tr>
<td>--kp_hmac_inputkey</td>
<td>File having the HMAC key for generating the NVS encryption keys</td>
</tr>
<tr>
<td><strong>KP_HMAC_KEYFILE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>KP_HMAC_INPUTKEY</strong></td>
<td></td>
</tr>
</tbody>
</table>

You can run the utility to generate only encryption keys using the command below:

```bash
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`. To provide custom target filename use the `--keyfile` argument.

For generating encryption key for the HMAC-based scheme, the following commands can be used:

- Generate the HMAC key and the NVS encryption keys:

```bash
python mfg_gen.py generate-key --key_protect_hmac --kp_hmac_keygen
```

Note: Encryption key of the format `<outdir>/keys/keys-<timestamp>.bin` and HMAC key of the format `<outdir>/keys/hmac-keys-<timestamp>.bin` are created.

- Generate the NVS encryption keys, given the HMAC-key:

```bash
python mfg_gen.py generate-key --key_protect_hmac --kp_hmac_inputkey testdata/->sample_hmac_key.bin
```

Note: You can provide the custom filename for the HMAC key as well as the encryption key as a parameter.

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:
Chapter 2. API Reference

- **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)

### 2.8.3 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 the `esp_partition` API. 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_partition()` 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 508,000 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 replaces the old value and data type with the value and data type specified by a write operation.

A data type check is performed when reading a value. An error is returned if the data type expected by read operation does not match the data type of entry found for the key provided.

**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. Furthermore, there can be no more than 254 different namespaces in one NVS partition. Namespace name is specified in the `nvs_open()` or `nvs_open_from_partition` 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.

**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()` creates an opaque handle, which is used in subsequent calls to the `nvs_entry_next()` and `nvs_entry_info()` functions.
- `nvs_entry_next()` advances an iterator to the next key-value pair.
- `nvs_entry_info()` returns information about each key-value pair.

In general, all iterators obtained via `nvs_entry_find()` have to be released using `nvs_release_iterator()`, which also tolerates NULL iterators.

`nvs_entry_find()` and `nvs_entry_next()` set the given iterator to NULL or a valid iterator in all cases except a parameter error occurred (i.e., return `ESP_ERR_NVS_NOT_FOUND`). In case of a parameter error, the given iterator will not be modified. Hence, it is best practice to initialize the iterator to NULL before calling `nvs_entry_find()` to avoid complicated error checking before releasing the iterator.

**Security, Tampering, and Robustness** NVS is not directly compatible with the ESP32-S2 flash encryption system. However, data can still be stored in encrypted form if NVS encryption is used together with ESP32-S2 flash encryption or with the help of the HMAC peripheral. 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.

**NVS Encryption**

Please refer to the `NVS Encryption` guide for more details.

**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`.

Instead of calling the `nvs_partition_gen.py` tool manually, the creation of the partition binary files can also be done directly from CMake using the function `nvs_create_partition_image`:

```
nvs_create_partition_image(<partition> <csv> [FLASH_IN_PROJECT] [DEPENDS dep dep...
---dep ...])
```

### Positional Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition</td>
<td>Name of the NVS partition</td>
</tr>
<tr>
<td>csv</td>
<td>Path to CSV file to parse</td>
</tr>
</tbody>
</table>

### Optional Arguments:

Submit Document Feedback
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLASH_IN_PROJECT</td>
<td>Name of the NVS partition</td>
</tr>
<tr>
<td>DEPENDS</td>
<td>Specify files on which the command depends</td>
</tr>
</tbody>
</table>

If `FLASH_IN_PROJECT` is not specified, the image will still be generated, but you will have to flash it manually using `idf.py <partition>-flash` (e.g., if your partition name is `nvs`, then use `idf.py nvs-flash`). `nvs_create_partition_image` must be called from one of the component `CMakeLists.txt` files. Currently, only non-encrypted partitions are supported.

**Application Example**

You can find 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-S2 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-S2 module restarts.
  
  - value - tracks the number of the ESP32-S2 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.

**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 ESP32-S2 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 ESP32-S2 flash encryption, the entry size is 32 bytes. For integer types, an 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.

```
+-----------+-----------+-----------+-----------+-----------+-----------+
| Page 1    | Page 2    | Page 3    | Page 4    |-----------|
| Full      | Full      | Active    | Empty     |          <- states |
|           |           |           |           |          <- sequence numbers |
+-----------+-----------+-----------+-----------+-----------+
|           |           |           |           |          |
+-----------+-----------+-----------+-----------+-----------+
|           |           |           |           |          |
|           |           |           |           |          |
|           |           |           |           |          |
|           |           |           |           |          |
+-----------+-----------+-----------+-----------+-----------+
| v         | v         | v         | v         |          <- physical sectors |
|           |           |           |           |          |
+-----------+-----------+-----------+-----------+-----------+
| Sector 3  | Sector 0  | Sector 2  | Sector 1  |          |
|           |           |           |           |          |
+-----------+-----------+-----------+-----------+-----------+
```

Page header and entry state bitmap are always written to flash unencrypted. Entries are encrypted if flash encryption feature of ESP32-S2 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 0xfff 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.

**Eased (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.

![Blob Index Format](image)

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.

```
+----------------------------------| Entry describing namespace "wifi"
| NS-0 Type=uint8_t Key="wifi" Value=1 |
+----------------------------------| Entry describing namespace "pwm"
| NS-1 Type=uint32_t Key="channel" Value=6 | Key "channel" in namespace "wifi"
+----------------------------------| Entry describing namespace "pwm"
| NS-0 Type=uint8_t Key="pwm" Value=2 |
+----------------------------------| Key "channel" in namespace "pwm"
| NS-2 Type=uint16_t Key="channel" Value=20 |
```

**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 is 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.

**API Reference**

**Header File**

- `components/nvs_flash/include/nvs_flash.h`
- This header file can be included with:
  ```
  #include "nvs_flash.h"
  ```
- This header file is a part of the API provided by the `nvs_flash` component. To declare that your component depends on `nvs_flash`, add the following to your `CMakeLists.txt`:
  ```
  REQUIRES nvs_flash
  ```
- or
  ```
  PRIV_REQUIRES nvs_flash
  ```

**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.

  When “NVS_ENCRYPTION” is enabled in the menuconfig, this API enables the NVS encryption for the default NVS partition as follows
a. Read security configurations from the first NVS key partition listed in the partition table. (NVS key partition is any “data” type partition which has the subtype value set to “nvs_keys”)
b. If the NVS key partition obtained in the previous step is empty, generate and store new keys in that NVS key partition.
c. Internally call `nvs_flash_secure_init()` with the security configurations obtained/generated in the previous steps.

Post initialization NVS read/write APIs remain the same irrespective of NVS encryption.

**Returns**

- 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
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver
- error codes from `nvs_flash_read_security_cfg` API (when “NVS_ENCRYPTION” is enabled).
- error codes from `nvs_flash_generate_keys` API (when “NVS_ENCRYPTION” is enabled).
- error codes from `nvs_flash_secure_init_partition` API (when “NVS_ENCRYPTION” is enabled).

```c
esp_err_t nvs_flash_init_partition (const char *partition_label)
```

Initialize NVS flash storage for the specified partition.

**Parameters**

- `partition_label` -[in] Label of the partition. Must be no longer than 16 characters.

**Returns**

- 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
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

```c
esp_err_t nvs_flash_init_partition_ptr (const esp_partition_t *partition)
```

Initialize NVS flash storage for the partition specified by partition pointer.

**Parameters**

- `partition` -[in] pointer to a partition obtained by the ESP partition API.

**Returns**

- 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
- ESP_ERR_INVALID_ARG in case partition is NULL
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

```c
esp_err_t nvs_flash_deinit_partition (void)
```

Deinitialize NVS storage for the given NVS partition.

Default NVS partition is the partition with “nvs” label in the partition table.

**Returns**

- ESP_OK on success (storage was deinitialized)
- ESP_ERR_NVS_NOT_INITIALIZED if the storage was not initialized prior to this call

```c
esp_err_t nvs_flash_deinit_partition (const char *partition_label)
```

Deinitialize NVS storage for the given NVS partition.

**Parameters**

- `partition_label` -[in] Label of the partition

**Returns**

- ESP_OK on success
Chapter 2. API Reference

- ESP_ERR_NVS_NOT_INITIALIZED if the storage for given partition was not initialized prior to this call

```c
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.

**Returns**
- 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)

```c
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.

**Parameters**
- `part_name` - [in] Name (label) of the partition which should be erased

**Returns**
- 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)

```c
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.

**Parameters**
- `partition` - [in] pointer to a partition obtained by the ESP partition API.

**Returns**
- 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

```c
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.

**Parameters**
- `cfg` - [in] Security configuration (keys) to be used for NVS encryption/decryption.
  If `cfg` is NULL, no encryption is used.
Returns

- ESP_OK if storage has been initialized successfully.
- 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
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

```
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.

Parameters

- `partition_label` - [in] Label of the partition. Note that internally, a reference to passed value is kept and it should be accessible for future operations
- `cfg` - [in] Security configuration (keys) to be used for NVS encryption/decryption. If `cfg` is null, no encryption/decryption is used.

Returns

- ESP_OK if storage has been initialized successfully.
- 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
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- one of the error codes from the underlying flash storage driver

```
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.

Parameters

- `partition` - [in] Pointer to partition structure obtained using esp_partition_find_first or esp_partition_get. Must be non-NULL.
- `cfg` - [out] Pointer to nvs security configuration structure. Pointer must be non-NULL. Generated keys will be populated in this structure.

Returns

- ESP_OK, if `cfg` was read successfully;
- ESP_ERR_INVALID_ARG, if `partition` or `cfg` is NULL;
- or error codes from esp_partition_write/erase APIs.

```
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’.

```
esp_err_t nvs_flash_register_security_scheme(nvs_sec_scheme_t *scheme_cfg)
```

Registers the given security scheme for NVS encryption The scheme registered with `sec_scheme_id` by this...
API be used as the default security scheme for the “nvs” partition. Users will have to call this API explicitly in their application.

**Parameters**

 scheme_cfg - [in] Pointer to the security scheme configuration structure that the user (or the nvs_key_provider) wants to register.

**Returns**

- ESP_OK, if security scheme registration succeeds;
- ESP_ERR_INVALID_ARG, if scheme_cfg is NULL;
- ESP_FAIL, if security scheme registration fails

```c
nvs_sec_scheme_t *nvs_flash_get_default_security_scheme (void)
```

Fetch the configuration structure for the default active security scheme for NVS encryption.

**Returns**

- Pointer to the default active security scheme configuration (NULL if no scheme is registered yet i.e. active)

```c
esp_err_t nvs_flash_generate_keys_v2 (nvs_sec_scheme_t *scheme_cfg, nvs_sec_cfg_t *cfg)
```

Generate (and store) the NVS keys using the specified key-protection scheme.

**Parameters**

- scheme_cfg - [in] Security scheme specific configuration
- cfg - [out] Security configuration (encryption keys)

**Returns**

- ESP_OK, if cfg was populated successfully with generated encryption keys;
- ESP_ERR_INVALID_ARG, if scheme_cfg or cfg is NULL;
- ESP_FAIL, if the key generation process fails

```c
esp_err_t nvs_flash_read_security_cfg_v2 (nvs_sec_scheme_t *scheme_cfg, nvs_sec_cfg_t *cfg)
```

Read NVS security configuration set by the specified security scheme.

**Parameters**

- scheme_cfg - [in] Security scheme specific configuration
- cfg - [out] Security configuration (encryption keys)

**Returns**

- ESP_OK, if cfg was read successfully;
- ESP_ERR_INVALID_ARG, if scheme_cfg or cfg is NULL;
- ESP_FAIL, if the key reading process fails

**Structures**

```c
struct nvs_sec_cfg_t
```

- Key for encryption and decryption.

**Public Members**

```c
uint8_t eky[NVS_KEY_SIZE]
```

- XTS encryption and decryption key

```c
uint8_t tky[NVS_KEY_SIZE]
```

- XTS tweak key

**Public Members**

```c
struct nvs_sec_scheme_t
```

- NVS encryption: Security scheme configuration structure.
int scheme_id  
    Security Scheme ID (E.g. HMAC)

void *scheme_data  
    Scheme-specific data (E.g. eFuse block for HMAC-based key generation)

nvs_flash_generate_keys_t nvs_flash_key_gen  
    Callback for the nvs_flash_key_gen implementation

nvs_flash_read_cfg_t nvs_flash_read_cfg  
    Callback for the nvs_flash_read_keys implementation

Macros

NVS_KEY_SIZE

Type Definitions

typedef esp_err_t (*nvs_flash_generate_keys_t)(const void *scheme_data, nvs_sec_cfg_t *cfg)  
    Callback function prototype for generating the NVS encryption keys.

typedef esp_err_t (*nvs_flash_read_cfg_t)(const void *scheme_data, nvs_sec_cfg_t *cfg)  
    Callback function prototype for reading the NVS encryption keys.

Header File

- components/nvs_flash/include/nvs.h  
- This header file can be included with:

```c
#include "nvs.h"
```

- This header file is a part of the API provided by the nvs_flash component. To declare that your component depends on nvs_flash, add the following to your CMakeLists.txt:

```cmake
REQUIRES nvs_flash
```

or

```cmake
PRIV_REQUIRES nvs_flash
```

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. Regardless whether key-value pair is created or updated, function always requires at least one nvs available entry. See nvs_get_stats. After create type of operation, the number of available entries is decreased by one. After update type of operation, the number of available entries remains the same.

Parameters

- handle -[in] Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- key -[in] Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.
- value -[in] The value to set.
Chapter 2. API Reference

Returns
- ESP_OK if value was set successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- 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 nvss, provided that flash operation doesn’t fail again.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
esp_err_t nvs_set_str (nvs_handle_t handle, const char* key, const char* value)
```

set string for given key

Sets string value for the key. Function requires whole space for new data to be available as contiguous entries in same nvs page. Operation consumes 1 overhead entry and 1 entry per each 32 characters of new string including zero character to be set. In case of value update for existing key, entries occupied by the previous value and overhead entry are returned to the pool of available entries. Note that storage of long string values can fail due to fragmentation of nvs pages even if available_entries returned by nvs_get_stats suggests enough overall space available. Note that the underlying storage will not be updated until nvs_commit is called.

Parameters
- handle –[in] Handle obtained from nvs_open function. Handles that were opened read only cannot be used.
- key –[in] Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.
Chapter 2. API Reference

value -[in] 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.

Returns
- 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

```
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.

Parameters
- handle -[in] Handle obtained from nvs_open function.
- key -[in] Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) 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.

Returns
- ESP_OK if the value was retrieved successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- 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

```
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.
```

Espessif Systems  1292  Release master
Submit Document Feedback
Chapter 2. API Reference

```c
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.
```

```c
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.
```

```c
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.
```

```c
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.
```

```c
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 a 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);
```

Parameters

- **handle** - [in] Handle obtained from nvs_open function.
- **key** - [in] Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.
- **out_value** - [out] 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.
- **length** - [inout] 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.

Returns

- ESP_OK if the value was retrieved successfully
Chapter 2. API Reference

- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- 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

```c
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.

```c
esp_err_t nvs_open (const char* namespace_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.

**Parameters**

- `namespace_name` - [in] Namespace name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.
- `open_mode` - [in] 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_handle` - [out] If successful (return code is zero), handle will be returned in this argument.

**Returns**

- ESP_OK if storage handle was opened successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- 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
- ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
- ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is no space for a new entry or there are too many different namespaces (maximum allowed different namespaces: 254)
- ESP_ERR_NOT_ALLOWED if the NVS partition is read-only and mode is NVS_READWRITE
- ESP_ERR_INVALID_ARG if out_handle is equal to NULL
- other error codes from the underlying storage driver

```c
esp_err_t nvs_open_from_partition (const char*part_name, const char*namespace_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.

**Parameters**

- `part_name` - [in] Label (name) of the partition of interest for object read/write/erase
- `namespace_name` - [in] Namespace name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.
- `open_mode` - [in] 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_handle` - [out] If successful (return code is zero), handle will be returned in this argument.

**Returns**

- ESP_OK if storage handle was opened successfully
• ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
• 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 if namespace doesn’t exist yet and mode is NVS_READONLY
• ESP_ERR_NVS_INVALID_NAME if namespace name doesn’t satisfy constraints
• ESP_ERR_NO_MEM in case memory could not be allocated for the internal structures
• ESP_ERR_NVS_NOT_ENOUGH_SPACE if there is no space for a new entry or there are too many different namespaces (maximum allowed different namespaces: 254)
• ESP_ERR_NOT_ALLOWED if the NVS partition is read-only and mode is NVS_READWRITE
• ESP_ERR_INTEGRITY if out_handle is equal to NULL
• other error codes from the underlying storage driver

`esp_err_t nvs_set_blob(nvs_handle_t handle, const char* key, const void* value, size_t length)`

Sets variable length binary value for given key. Function uses 2 overhead and 1 entry per each 32 bytes of new data from the pool of available entries. See `nvs_get_stats`. In case of value update for existing key, space occupied by the existing value and 2 overhead entries are returned to the pool of available entries. Note that the underlying storage will not be updated until `nvs_commit` is called.

**Parameters**
- `handle` [-in] Handle obtained from `nvs_open` function. Handles that were opened read only cannot be used.
- `key` [-in] Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.
- `value` [-in] The value to set.
- `length` [-in] 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.

**Returns**
- ESP_OK if value was set successfully
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- 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

`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.

**Parameters**
- `handle` [-in] Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.
- `key` [-in] Key name. Maximum length is (NVS_KEY_NAME_MAX_SIZE-1) characters. Shouldn’t be empty.

**Returns**
- ESP_OK if erase operation was successful
- ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
- 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

\( \text{esp_err_t nvs_erase_all(nvs_handle_t handle)} \)

Erase all key-value pairs in a namespace.

Note that actual storage may not be updated until nvs_commit function is called.

**Parameters**  
handle [in] Storage handle obtained with nvs_open. Handles that were opened read only cannot be used.

**Returns**  
• ESP_OK if erase operation was successful
• ESP_FAIL if there is an internal error; most likely due to corrupted NVS partition (only if NVS assertion checks are disabled)
• 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

\( \text{esp_err_t nvs_commit(nvs_handle_t handle)} \)

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.

**Parameters**  
handle [in] Storage handle obtained with nvs_open. Handles that were opened read only cannot be used.

**Returns**  
• 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

\( \text{void nvs_close(nvs_handle_t handle)} \)

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**  
handle [in] Storage handle to close

\( \text{esp_err_t nvs_get_stats(const char *part_name, nvs_stats_t *nvs_stats)} \)

Fill structure nvs_stats_t. It provides info about memory used by NVS.

This function calculates the number of used entries, free entries, available entries, total entries and number of namespaces in partition.

```c
// Example of nvs_get_stats() to get overview of actual statistics of data...

const char *part_name = "partition_name";
const nvs_stats_t *nvs_stats;

nvs_stats_t nvs_stats;
if (nvs_get_stats(NULL, &nvs_stats) == ESP_OK) {
    printf("Count: UsedEntries = (%lu), FreeEntries = (%lu), AvailableEntries = (%lu), AllEntries = (%lu)\n",
           nvs_stats.used_entries, nvs_stats.free_entries, nvs_stats.available_entries, nvs_stats.total_entries);
}
```

**Parameters**  
• part_name [in] Partition name NVS in the partition table. If pass a NULL than will use NVS_DEFAULT_PART_NAME("nvs").
• nvs_stats [out] Returns filled structure nvs_stats_t. It provides info about used memory the partition.

**Returns**
• 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 is 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.

```c
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.

```c
// 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;
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;
}
```

**Parameters**

- **handle** - [in] Handle obtained from nvs_open function.
- **used_entries** - [out] Returns amount of used entries from a namespace.

**Returns**

- 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 is equal to NULL.
- Other error codes from the underlying storage driver. Return param used_entries will be filled 0.

```c
esp_err_t nvs_entry_find(const char*part_name,const char*namespace_name, nvs_type_t type,
                        nvs_iterator_t*output_iterator)
```

Create an iterator to enumerate NVS entries based on one or more parameters.

```c
// Example of listing all the key-value pairs of any type under specified partition and namespace
nvs_iterator_t it = NULL;
esp_err_t res = nvs_entry_find(<nvs_partition_name>, <namespace>, NVS_TYPE_--ANY, &it);
while(res == ESP_OK) {
    nvs_entry_info_t info;
    nvs_entry_info(it, &info); // Can omit error check if parameters are guaranteed to be non NULL
}
```

(continues on next page)
printf("key '%s', type '%d' \n", info.key, info.type);
res = nvs_entry_next(&it);
}
nvs_release_iterator(it);

Parameters

- part_name - [in] Partition name
- namespace_name - [in] Set this value if looking for entries with a specific namespace. Pass NULL otherwise.
- type - [in] One of nvs_type_t values.
- output_iterator - [out] Set to a valid iterator to enumerate all the entries found. Set to NULL if no entry for specified criteria was found. If any other error except ESP_ERR_INVALID_ARG occurs, output_iterator is NULL, too. If ESP_ERR_INVALID_ARG occurs, output_iterator is not changed. If a valid iterator is obtained through this function, it has to be released using nvs_release_iterator when not used any more, unless ESP_ERR_INVALID_ARG is returned.

Returns

- ESP_OK if no internal error or programming error occurred.
- ESP_ERR_NVS_NOT_FOUND if no element of specified criteria has been found.
- ESP_ERR_NO_MEM if memory has been exhausted during allocation of internal structures.
- ESP_ERR_INVALID_ARG if any of the parameters is NULL. Note: don’t release output_iterator in case ESP_ERR_INVALID_ARG has been returned.

```
esp_err_t nvs_entry_find_in_handle(nvs_handle_t handle, nvs_type_t type, nvs_iterator_t *output_iterator)
```

Create an iterator to enumerate NVS entries based on a handle and type.

```
// Example of listing all the key-value pairs of any type under specified handle (which defines a partition and namespace)
nvs_iterator_t it = NULL;
esp_err_t res = nvs_entry_find_in_handle<nvs_handle>, NVS_TYPE_ANY, &it);
while(res == ESP_OK) {
  nvs_entry_info_t info;
  nvs_entry_info(it, &info); // Can omit error check if parameters are guaranteed to be non-NUL
  printf("key '%s', type '%d' \n", info.key, info.type);
  res = nvs_entry_next(&it);
}
nvs_release_iterator(it);
```

Parameters

- handle - [in] Handle obtained from nvs_open function.
- type - [in] One of nvs_type_t values.
- output_iterator - [out] Set to a valid iterator to enumerate all the entries found. Set to NULL if no entry for specified criteria was found. If any other error except ESP_ERR_INVALID_ARG occurs, output_iterator is NULL, too. If ESP_ERR_INVALID_ARG occurs, output_iterator is not changed. If a valid iterator is obtained through this function, it has to be released using nvs_release_iterator when not used any more, unless ESP_ERR_INVALID_ARG is returned.

Returns

- ESP_OK if no internal error or programming error occurred.
- ESP_ERR_NVS_NOT_FOUND if no element of specified criteria has been found.
• ESP_ERR_NO_MEM if memory has been exhausted during allocation of internal structures.
• ESP_ERR_NVS_INVALID_HANDLE if unknown handle was specified.
• ESP_ERR_INVALID_ARG if output_iterator parameter is NULL. Note: don’t release output_iterator in case ESP_ERR_INVALID_ARG has been returned

```
esp_err_t nvs_entry_next (nvs_iterator_t *iterator)
```

Advances the iterator to next item matching the iterator criteria.

Note that any copies of the iterator will be invalid after this call.

**Parameters**
- `iterator` - [inout] Iterator obtained from nvs_entry_find or nvs_entry_find_in_handle function. Must be non-NULL. If any error except ESP_ERR_INVALID_ARG occurs, iterator is set to NULL. If ESP_ERR_INVALID_ARG occurs, iterator is not changed.

**Returns**
- ESP_OK if no internal error or programming error occurred.
- ESP_ERR_NVS_NOT_FOUND if no next element matching the iterator criteria.
- ESP_ERR_INVALID_ARG if iterator is NULL.
- Possibly other errors in the future for internal programming or flash errors.

```
esp_err_t nvs_entry_info (const 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**
- `iterator` - [in] Iterator obtained from nvs_entry_find or nvs_entry_find_in_handle function. Must be non-NULL.
- `out_info` - [out] Structure to which entry information is copied.

**Returns**
- ESP_OK if all parameters are valid; current iterator data has been written to out_info
- ESP_ERR_INVALID_ARG if one of the parameters is NULL.

```
void nvs_release_iterator (nvs_iterator_t iterator)
```

Release iterator.

**Parameters**
- `iterator` - [in] Release iterator obtained from nvs_entry_find or nvs_entry_find_in_handle or nvs_entry_next 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[NVS_NS_NAME_MAX_SIZE]
```

- Namespace to which key-value belong

```
char key[NVS_KEY_NAME_MAX_SIZE]
```

- Key of stored key-value pair

```
nvs_type_t type
```

- Type of stored key-value pair

```
struct nvs_stats_t
```


Chapter 2. API Reference

**Note:** Info about storage space NVS.

**Public Members**

- `size_t used_entries`
  Number of used entries.

- `size_t free_entries`
  Number of free entries. It includes also reserved entries.

- `size_t available_entries`
  Number of entries available for data storage.

- `size_t total_entries`
  Number of all entries.

- `size_t namespace_count`
  Number of namespaces.

**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`
  A requested entry couldn’t be found or 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**

Value doesn’t fit into the entry or 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
Chapter 2. API Reference

ESP_ERR_NVS_WRONG_ENCRYPTION
NVS partition is marked as encrypted with generic flash encryption. This is forbidden since the NVS encryption works differently.

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)

NVS_KEY_NAME_MAX_SIZE
Maximum length of NVS key name (including null terminator)

NVS_NS_NAME_MAX_SIZE
Maximum length of NVS namespace name (including 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:

enumerator NVS_READONLY
Read only

enumerator NVS_READWRITE
Read and write

enum nvs_type_t
Types of variables.
Values:

enumerator NVS_TYPE_U8
Type uint8_t
**Chapter 2. API Reference**

enumerator **NVS_TYPE_I8**
Type int8_t

enumerator **NVS_TYPE_U16**
Type uint16_t

enumerator **NVS_TYPE_I16**
Type int16_t

enumerator **NVS_TYPE_U32**
Type uint32_t

enumerator **NVS_TYPE_I32**
Type int32_t

enumerator **NVS_TYPE_U64**
Type uint64_t

enumerator **NVS_TYPE_I64**
Type int64_t

enumerator **NVS_TYPE_STR**
Type string

enumerator **NVS_TYPE_BLOB**
Type blob

enumerator **NVS_TYPE_ANY**
Must be last

### 2.8.4 NVS Encryption

**Overview**

This guide provides an overview of the NVS encryption feature. NVS encryption helps to achieve secure storage on the device flash memory.

Data stored in NVS partitions can be encrypted using XTS-AES 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.

NVS encryption can be facilitated by enabling `CONFIG_NVS_ENCRYPTION` and `CONFIG_NVS_SEC_KEY_PROTECTION_SCHEME` depending on the scheme to be used.
NVS Encryption: Flash Encryption-Based Scheme

In this scheme, the keys required for NVS encryption are stored in yet another partition, which is protected using Flash Encryption. Therefore, enabling Flash Encryption becomes a prerequisite for NVS encryption here.

NVS encryption is enabled by default when Flash Encryption is enabled. This is done because Wi-Fi driver stores credentials (like SSID and passphrase) in the default NVS partition. It is important to encrypt them as default choice if platform level encryption is already enabled.

For using NVS encryption using this scheme, the partition table must contain the NVS Key Partition. Two partition tables containing the NVS Key Partition are provided for NVS encryption under the partition table option (menuconfig>Partition Table). They can be selected with the project configuration menu (idf.py menuconfig). Please refer to the example security/flash_encryption for how to configure and use the NVS encryption feature.

NVS Key Partition An application requiring NVS encryption support (using the Flash Encryption-based scheme) needs to be compiled with a key-partition of the type data and subtype key. This partition should be marked as encrypted and its size should be the minimum partition size (4 KB). Refer to Partition Tables for more details. Two additional partition tables which contain the NVS Key Partition are provided under the partition table option (menuconfig>Partition Table). They can be directly used for NVS encryption. The structure of these partitions is depicted below:

+-------------------------------+--------------------------+-----------------------------+
| XTS encryption key (32) | +--------------------------+-----------------------------+
| +--------------------------+ XTS tweak key (32) | +--------------------------+
| +--------------------------+ CRC32 (4) | +--------------------------+

The XTS encryption keys in the NVS Key Partition can be generated in one of the following two ways.

Generate the keys on ESP32-S2 chip itself

- When NVS encryption is enabled, the nvs_flash_init() API function can be used to initialize the encrypted default NVS partition. The API function internally generates the XTS encryption keys on the ESP chip. The API function finds the first NVS Key Partition.
- Then the API function automatically generates and stores the NVS keys in that partition by making use of the nvs_flash_generate_keys() API function provided by nvs_flash/include/nvs_flash.h. New keys are generated and stored only when the respective key partition is empty. The same key partition can then be used to read the security configurations for initializing a custom encrypted NVS partition with help of nvs_flash_secure_init_partition().
- The API functions nvs_flash_secure_init() and nvs_flash_secure_init_partition() do not generate the keys internally. When these API functions are used for initializing encrypted NVS partitions, the keys can be generated after startup using the nvs_flash_generate_keys() API function provided by nvs_flash.h. The API function then writes those keys onto the key-partition in encrypted form.

Note: Please note that nvs_keys partition must be completely erased before you start the application in this approach. Otherwise the application may generate the ESP_ERR_NVS_CORRUPT_KEY_PART error code assuming that nvs_keys partition is not empty and contains malformatted data. You can use the following command for this:

```
parttool.py --port PORT --partition-table-file=PARTITION_TABLE_FILE --partition-table-offset PARTITION_TABLE_OFFSET erase_partition --partition-type=data --partition-subtype=nvs_keys
```

Use a pre-generated NVS key partition
This option will be required by the user when keys in the NVS Key Partition are not generated by the application. The NVS Key Partition containing the XTS encryption keys can be generated with the help of NVS Partition Generator Utility. Then the user can store the pre-generated key partition on the flash with help of the following two commands:

1. Build and flash the partition table
   ```bash
   idf.py partition-table partition-table-flash
   ```

2. Store the keys in the NVS Key Partition (on the flash) with the help of parttool.py (see Partition Tool section in partition-tables for more details)
   ```bash
   parttool.py --port PORT --partition-table-offset PARTITION_TABLE_OFFSET_ 
   --write_partition --partition-name="name of nvs_key partition" --input_ 
   --NVS_KEY_PARTITION_FILE
   ```

**Note:** If the device is encrypted in flash encryption development mode and you want to renew the NVS key partition, you need to tell parttool.py to encrypt the NVS key partition and you also need to give it a pointer to the unencrypted partition table in your build directory (build/partition_table) since the partition table on the device is encrypted, too. You can use the following command:

```bash
parttool.py --esptool-write-args encrypt --port PORT --partition-table- 
--file=PARTITION_TABLE_FILE --partition-table-offset PARTITION_TABLE_ 
--OFFSET write_partition --partition-name="name of nvs_key partition" -- 
--input NVS_KEY_PARTITION_FILE
```

Since the key partition is marked as encrypted and Flash Encryption is enabled, the bootloader will encrypt this partition using flash encryption key on the first boot.

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 the correct key-partition and keys for encryption or decryption.

**NVS Encryption: HMAC Peripheral-Based Scheme**

In this scheme, the XTS keys required for NVS encryption are derived from an HMAC key programmed in eFuse with the purpose `esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_UP`. Since the encryption keys are derived at runtime, they are not stored anywhere in the flash. Thus, this feature does not require a separate NVS Key Partition.

**Note:** This scheme enables us to achieve secure storage on ESP32-S2 without enabling flash encryption.

**Important:** Please note that this scheme uses one eFuse block for storing the HMAC key required for deriving the encryption keys.

- When NVS encryption is enabled, the `nvs_flash_init()` API function can be used to initialize the encrypted default NVS partition. The API function first checks whether an HMAC key is present at `CONFIG_NVS_SEC_HMAC_EFUSE_KEY_ID`.

**Note:** The valid range for the config `CONFIG_NVS_SEC_HMAC_EFUSE_KEY_ID` is from 0 (`hmac_key_id_t::HMAC_KEY0`) to 5 (`hmac_key_id_t::HMAC_KEY5`). By default, the config is set to 6 (`hmac_key_id_t::HMAC_KEY_MAX`), which have to be configured before building the user application.
Chapter 2. API Reference

• If no key is found, a key is generated internally and stored at the eFuse block specified at CON-
  FIG_NVS_SEC_HMAC_EFUSE_KEY_ID.
• If a key is found with the purpose `esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_UP`, the same is used for the derivation of the XTS encryption keys.
• If the specified eFuse block is found to be occupied with a key with a purpose other than `esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_UP`, an error is thrown.
• The API `nvs_flash_init()` then automatically generates the NVS keys on demand by using the `nvs_flash_generate_keys_v2()` API function provided by the `nvs_flash#include/nvs_flash.h`. The same keys can also be used to read the security configurations (see `nvs_flash_read_security_cfg_v2()`) for initializing a custom encrypted NVS partition with help of `nvs_flash_secure_init_partition()`.
• The API functions `nvs_flash_secure_init()` and `nvs_flash_secure_init_partition()` do not generate the keys internally. When these API functions are used for initializing encrypted NVS partitions, the keys can be generated after startup using the `nvs_flash_generate_keys_v2()` API function or take and populate the NVS security configuration structure `nvs_sec_cfg_t` with `nvs_flash_read_security_cfg_v2()` and feed them into the above APIs.

Note: Users can program their own HMAC key in eFuse block beforehand by using the following command:

espefuse.py -p PORT burn_key <BLOCK_KEYN> <hmac_key_file.bin> HMAC_UP

### Encrypted Read/Write

The same NVS APIs `nvs_get_*` or `nvs_set_*` can be used for reading of, and writing to an encrypted NVS partition as well.

#### Encrypt the default NVS partition

• To enable encryption for the default NVS partition, no additional step is necessary. When `CONFIG_NVS_ENCRYPTION` is enabled, the `nvs_flash_init()` API function internally performs some additional steps to enable encryption for the default NVS partition depending on the scheme being used (set by `CONFIG_NVS_SEC_KEY_PROTECTION_SCHEME`).
• For the flash encryption-based scheme, the first `NVS Key Partition` found is used to generate the encryption keys while for the HMAC one, keys are generated using the HMAC key burnt in eFuse at `CONFIG_NVS_SEC_HMAC_EFUSE_KEY_ID` (refer to the API documentation for more details).

Alternatively, `nvs_flash_secure_init()` API function can also be used to enable encryption for the default NVS partition.

#### Encrypt a custom NVS partition

• To enable encryption for a custom NVS partition, `nvs_flash_secure_init_partition()` API function is used instead of `nvs_flash_init_partition()`.
• When `nvs_flash_secure_init()` and `nvs_flash_secure_init_partition()` API functions are used, the applications are expected to follow the steps below in order to perform NVS read/write operations with encryption enabled:
  1. Populate the NVS security configuration structure `nvs_sec_cfg_t`
     - For the Flash Encryption-based scheme
       * Find key partition and NVS data partition using `esp_partition_find*` API functions.
       * Populate the `nvs_sec_cfg_t` struct using the `nvs_flash_read_security_cfg()` or `nvs_flash_generate_keys()` API functions.
     - For the HMAC-based scheme
       * Set the scheme-specific config data with `nvs_sec_config_hmac_t` and register the HMAC-based scheme with the API `nvs_sec_provider_register_hmac()` which will also populate the scheme-specific handle (see `nvs_sec_scheme_t`).
* Populate the `nvs_sec_cfg_t` struct using the `nvs_flash_read_security_cfg_v2()` or `nvs_flash_generate_keys_v2()` API functions.

```c
nvs_sec_cfg_t cfg = {}; 
nvs_sec_scheme_t *sec_scheme_handle = NULL;

nvs_sec_config_hmac_t sec_scheme_cfg = {}; 
hmac_key_id_t hmac_key = HMAC_KEY0;
sec_scheme_cfg.hmac_key_id = hmac_key;

ret = nvs_sec_provider_register_hmac(&sec_scheme_cfg, &sec_scheme_handle);
if (ret != ESP_OK) {
    return ret;
}
ret = nvs_flash_read_security_cfg_v2(sec_scheme_handle, &cfg);
if (ret != ESP_OK) {
    if (ret == ESP_ERR_NVSEC_SEC_HMAC_KEY_NOT_FOUND) {
        ret = nvs_flash_generate_keys_v2(&sec_scheme_handle, &cfg);
        if (ret != ESP_OK) {
            ESP_LOGE(TAG, "Failed to generate NVS encr-keys!");
            return ret;
        }
    }
    ESP_LOGE(TAG, "Failed to read NVS security cfg!");
    return ret;
}
```

2. Initialise NVS flash partition using the `nvs_flash_secure_init()` or `nvs_flash_secure_init_partition()` API functions.
3. Open a namespace using the `nvs_open()` or `nvs_open_from_partition()` API functions.
4. Perform NVS read/write operations using `nvs_get_*` or `nvs_set_*`.
5. Deinitialise an NVS partition using `nvs_flash_deinit()`.

**Note:** While using the HMAC-based scheme, the above workflow can be used without enabling any of the config options for NVS encryption - `CONFIG_NVS_ENCRYPTION`, `CONFIG_NVS_SEC_KEY_PROTECTION_SCHEME` -> `CONFIG_NVS_SEC_KEY_PROTECT_USING_HMAC` and `CONFIG_NVS_SEC_HMAC_EFUSE_KEY_ID` to encrypt the default as well as custom NVS partitions with `nvs_flash_secure_init()` API.

---

**NVS Security Provider**

The component `nvs_sec_provider` stores all the implementation-specific code for the NVS encryption schemes and would also accommodate any future schemes. This component acts as an interface to the `nvs_flash` component for the handling of encryption keys. `nvs_sec_provider` has a configuration menu of its own, based on which the selected security scheme and the corresponding settings are registered for the `nvs_flash` component.

This component offers factory functions with which a particular security scheme can be registered without having to worry about the APIs to generate and read the encryption keys (e.g., `nvs_sec_provider_register_hmac()`). Refer to the `security/nvs_encryption_hmac` example for API usage.

**API Reference**

**Header File**

- `components/nvs_sec_provider/include/nvs_sec_provider.h`
- This header file can be included with:
Chapter 2. API Reference

```c
#include "nvs_sec_provider.h"
```

- This header file is a part of the API provided by the `nvs_sec_provider` component. To declare that your component depends on `nvs_sec_provider`, add the following to your CMakeLists.txt:

  ```cmake
  REQUIRE(nvs_sec_provider)
  ```
  
or

  ```cmake
  PRIV_REQUIRES(nvs_sec_provider)
  ```

### Functions

**`esp_err_t nvs_sec_provider_register_flash_enc(const nvs_sec_config_flash_enc_t *sec_scheme_cfg, nvs_sec_scheme_t **sec_scheme_handle_out)`**

Register the Flash-Encryption based scheme for NVS Encryption.

**Parameters**
- `sec_scheme_cfg` - [in] Security scheme specific configuration data
- `sec_scheme_handle_out` - [out] Security scheme specific configuration handle

**Returns**
- `ESP_OK`, if `sec_scheme_handle_out` was populated successfully with the scheme configuration;
- `ESP_ERR_INVALID_ARG`, if `scheme_cfg_hmac` is NULL;
- `ESP_ERR_NO_MEM`, No memory for the scheme-specific handle `sec_scheme_handle_out`
- `ESP_ERR_NOT_FOUND`, if no `nvs_keys` partition is found

**`esp_err_t nvs_sec_provider_register_hmac(const nvs_sec_config_hmac_t *sec_scheme_cfg, nvs_sec_scheme_t **sec_scheme_handle_out)`**

Register the HMAC-based scheme for NVS Encryption.

**Parameters**
- `sec_scheme_cfg` - [in] Security scheme specific configuration data
- `sec_scheme_handle_out` - [out] Security scheme specific configuration handle

**Returns**
- `ESP_OK`, if `sec_scheme_handle_out` was populated successfully with the scheme configuration;
- `ESP_ERR_INVALID_ARG`, if `scheme_cfg_hmac` is NULL;
- `ESP_ERR_NO_MEM`, No memory for the scheme-specific handle `sec_scheme_handle_out`

**`esp_err_t nvs_sec_provider_deregister(nvs_sec_scheme_t *sec_scheme_handle)`**

Deregister the NVS encryption scheme registered with the given handle.

**Parameters**
- `sec_scheme_handle` - [in] Security scheme specific configuration handle

**Returns**
- `ESP_OK`, if the scheme registered with `sec_scheme_handle` was deregistered successfully
- `ESP_ERR_INVALID_ARG`, if `sec_scheme_handle` is NULL;

### Structures

**`struct nvs_sec_config_flash_enc_t`**

Flash encryption-based scheme specific configuration data.

### Public Members
const esp_partition_t *nvs_keys_part

Partition of subtype nvs_keys holding the NVS encryption keys

struct nvs_sec_config_hmac_t

HMAC-based scheme specific configuration data.

**Public Members**

hmac_key_id_t hmac_key_id

HMAC Key ID used for generating the NVS encryption keys

**Macros**

ESP_ERR_NVS_SEC_BASE

Starting number of error codes

ESP_ERR_NVS_SEC_HMAC_KEY_NOT_FOUND

HMAC Key required to generate the NVS encryption keys not found

ESP_ERR_NVS_SEC_HMAC_KEY_BLK_ALREADY_USED

Provided eFuse block for HMAC key generation is already in use

ESP_ERR_NVS_SEC_HMAC_KEY_GENERATION_FAILED

Failed to generate/write the HMAC key to eFuse

ESP_ERR_NVS_SEC_HMAC_XTS_KEYS_DERIV_FAILED

Failed to derive the NVS encryption keys based on the HMAC-based scheme

NVS_SEC_PROVIDER_CFG_FLASH_ENC_DEFAULT()

Helper for populating the Flash encryption-based scheme specific configuration data.

NVS_SEC_PROVIDER_CFG_HMAC_DEFAULT()

Helper for populating the HMAC-based scheme specific configuration data.

**Enumerations**

enum nvs_sec_scheme_id_t

NVS Encryption Keys Protection Scheme.

**Values:**

enumerator NVS_SEC_SCHHEME_FLASH_ENC

Protect NVS encryption keys using Flash Encryption

enumerator NVS_SEC_SCHHEME_HMAC

Protect NVS encryption keys using HMAC peripheral

enumerator NVS_SEC_SCHHEME_MAX

**2.8.5 NVS Partition Generator Utility**
Introduction

The utility `nvs_flash/nvs_partition_generator/nvs_partition_gen.py` creates a binary file, compatible with the NVS architecture defined in *Non-Volatile Storage Library*, based on the key-value pairs provided in a CSV file.

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

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 describes each of these parameters.

<table>
<thead>
<tr>
<th>No.</th>
<th>Parameter</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Key</td>
<td>Key of the data. The data can be accessed later from an application using this key.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Type</td>
<td>Supported values are <em>file</em>, <em>data</em>, and <em>namespace</em>.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Encoding</td>
<td>Supported values are: u8, i8, u16, i16, u32, i32, u64, i64, string, hex2bin, base64, and binary. This specifies how actual data values are encoded in the resulting binary file. The difference between the <em>string</em> and <em>binary</em> encoding is that <em>string</em> data is terminated with a NULL character, whereas <em>binary</em> data is not. As of now, for the <em>file</em> type, only hex2bin, base64, <em>string</em>, and <em>binary</em> encoding is supported. Encoding and Value cells for the <em>namespace</em> field type should be empty. Encoding and Value of <em>namespace</em> are fixed and are not configurable. Any values in these cells are ignored.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Value</td>
<td>Data value</td>
<td></td>
</tr>
</tbody>
</table>

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 the older format, the utility provides an option to disable this feature.

**Encryption-Decryption Support**

The NVS Partition Generator utility also allows you to create an encrypted binary file and decrypt an encrypted one. The utility uses the XTS-AES 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:**

<table>
<thead>
<tr>
<th>No.</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-h/--help</td>
<td>Show the help message and exit</td>
</tr>
</tbody>
</table>

**Commands:**

Run `nvs_partition_gen.py {command} -h` for additional help

<table>
<thead>
<tr>
<th>No.</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>generate</td>
<td>Generate NVS partition</td>
</tr>
<tr>
<td>2</td>
<td>generate-key</td>
<td>Generate keys for encryption</td>
</tr>
<tr>
<td>3</td>
<td>encrypt</td>
<td>Generate NVS encrypted partition</td>
</tr>
<tr>
<td>4</td>
<td>decrypt</td>
<td>Decrypt NVS encrypted partition</td>
</tr>
</tbody>
</table>

**Generate NVS Partition (Default)**

**Usage:**

```
python nvs_partition_gen.py generate [-h] [--version {1,2}] [--outdir OUTDIR]--input input output size
```

**Positional Arguments:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Path to CSV file to parse</td>
</tr>
<tr>
<td>output</td>
<td>Path to output NVS binary file</td>
</tr>
<tr>
<td>size</td>
<td>Size of NVS partition in bytes (must be multiple of 4096)</td>
</tr>
</tbody>
</table>
Chapter 2. API Reference

Optional Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h/--help</td>
<td>Show the help message and exit</td>
</tr>
<tr>
<td>--version (1,2)</td>
<td>Set multipage blob version (Default: Version 2)</td>
</tr>
<tr>
<td></td>
<td>Version 1 - Multipage blob support disabled</td>
</tr>
<tr>
<td></td>
<td>Version 2 - Multipage blob support enabled</td>
</tr>
<tr>
<td>--outdir OUTDIR</td>
<td>Output directory to store file created (Default: current directory)</td>
</tr>
</tbody>
</table>

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
```

Generate Encryption Keys Partition  Usage:

```
python nvs_partition_gen.py generate-key [-h] [--key_protect_hmac] [--kp_hmac_keygen]

--kp_hmac_inputkey KP_HMAC_INPUTKEY

--kp_hmac_keyfile KP_HMAC_KEYFILE

--keyfile KEYFILE

--outdir OUTDIR
```

Optional Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h/--help</td>
<td>Show the help message and exit</td>
</tr>
<tr>
<td>--keyfile KEYFILE</td>
<td>Path to output encryption keys file</td>
</tr>
<tr>
<td>--outdir OUTDIR</td>
<td>Output directory to store files created. (Default: current directory)</td>
</tr>
</tbody>
</table>

Optional Arguments (HMAC scheme-specific):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--key_protect_hmac</td>
<td>If set, the NVS encryption key protection scheme based on HMAC peripheral is used; else the default scheme based on flash encryption is used</td>
</tr>
<tr>
<td>--kp_hmac_keygen</td>
<td>Generate the HMAC key for HMAC-based encryption scheme</td>
</tr>
<tr>
<td>--kp_hmac_keyfile KP_HMAC_KEYFILE</td>
<td>Path to output the HMAC key file</td>
</tr>
<tr>
<td>--kp_hmac_inputkey KP_HMAC_INPUTKEY</td>
<td>File having the HMAC key for generating the NVS encryption keys</td>
</tr>
</tbody>
</table>

You can run the utility to generate only the encryption key partition using the command below:

```
python nvs_partition_gen.py generate-key
```

For generating encryption key for the HMAC-based scheme, the following commands can be used:

- Generate the HMAC key and the NVS encryption keys:
  ```
  python nvs_partition_gen.py generate-key --key_protect_hmac --kp_hmac_keygen
  ```

**Note:** Encryption key of the format `<outdir>/keys/keys-<timestamp>.bin` and HMAC key of the format `<outdir>/keys/hmac-keys-<timestamp>.bin` are created.

- Generate the NVS encryption keys, given the HMAC key:
**Generate Encrypted NVS Partition**  
**Usage:**
```
python nvs_partition_gen.py encrypt [-h] [--version {1,2}] [--keygen]  
    [--keyfile KEYFILE] [--inputkey INPUTKEY] [--key_protect_hmac]  
    [--kp_hmac_keygen] [--kp_hmac_keyfile KP_HMAC_KEYFILE]  
    [--kp_hmac_inputkey KP_HMAC_INPUTKEY]  
```

**Positional Arguments:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Path to CSV file to parse</td>
</tr>
<tr>
<td>output</td>
<td>Path to output NVS binary file</td>
</tr>
<tr>
<td>size</td>
<td>Size of NVS partition in bytes (must be multiple of 4096)</td>
</tr>
</tbody>
</table>

**Optional Arguments:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h/ --help</td>
<td>Show the help message and exit</td>
</tr>
<tr>
<td>--version {1,2}</td>
<td>Set multipage blob version (Default: Version 2)</td>
</tr>
<tr>
<td></td>
<td>Version 1 - Multipage blob support disabled</td>
</tr>
<tr>
<td></td>
<td>Version 2 - Multipage blob support enabled</td>
</tr>
<tr>
<td>--keygen</td>
<td>Generates key for encrypting NVS partition</td>
</tr>
<tr>
<td>--keyfile KEYFILE</td>
<td>Path to output encryption keys file</td>
</tr>
<tr>
<td>--inputkey INPUTKEY</td>
<td>File having key for encrypting NVS partition</td>
</tr>
<tr>
<td>--outdir OUTDIR</td>
<td>Output directory to store file created (Default: current directory)</td>
</tr>
</tbody>
</table>

**Optional Arguments (HMAC scheme-specific):**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--key_protect_hmac</td>
<td>If set, the NVS encryption key protection scheme based on HMAC peripheral is used; else the default scheme based on flash encryption is used</td>
</tr>
<tr>
<td>--kp_hmac_keygen</td>
<td>Generate the HMAC key for HMAC-based encryption scheme</td>
</tr>
<tr>
<td>--kp_hmac_keyfile KP_HMAC_KEYFILE</td>
<td>Path to output HMAC key file</td>
</tr>
<tr>
<td>--kp_hmac_inputkey KP_HMAC_INPUTKEY</td>
<td>File having the HMAC key for generating the NVS encryption keys</td>
</tr>
</tbody>
</table>

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:
Chapter 2. API Reference

```python
python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
--0x3000 --keygen
```

**Note:** Encryption key of the format `<outdir>/keys/keys-<timestamp>.bin` is created.

- To generate an encrypted partition using the HMAC-based scheme, the above command can be used along with some additional parameters.
  - Encrypt by allowing the utility to generate encryption keys and the HMAC-key:
    ```python
    python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
    --0x3000 --keygen --key_protect_hmac --kp_hmac_keygen
    ```
    **Note:** Encryption key of the format `<outdir>/keys/keys-<timestamp>.bin` and HMAC key of the format `<outdir>/keys/hmac-keys-<timestamp>.bin` are created.
  - Encrypt by allowing the utility to generate encryption keys with user-provided HMAC-key:
    ```python
    python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
    --0x3000 --keygen --key_protect_hmac --kp_hmac_inputkey testdata/sample_hmac_key.bin
    ```
    **Note:** You can provide the custom filename for the HMAC key as well as the encryption key as a parameter.
  - Encrypt by allowing the utility to generate encryption keys and store it in provided custom filename:
    ```python
    python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
    --0x3000 --keygen --keyfile sample_keys.bin
    ```
    **Note:**
    - Encryption key of the format `<outdir>/keys/sample_keys.bin` is created.
    - 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
    python nvs_partition_gen.py encrypt sample_singlepage_blob.csv sample_encr.bin
    --0x3000 --inputkey sample_keys.bin
    ```

**Decrypt Encrypted NVS Partition**

**Usage:**

```python
python nvs_partition_gen.py decrypt [-h] [--outdir OUTDIR] input key output
```

**Positional Arguments:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Path to encrypted NVS partition file to parse</td>
</tr>
<tr>
<td>key</td>
<td>Path to file having keys for decryption</td>
</tr>
<tr>
<td>output</td>
<td>Path to output decrypted binary file</td>
</tr>
</tbody>
</table>

**Optional Arguments:**
### Chapter 2. API Reference

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code>/<code>--help</code></td>
<td>Show the help message and exit</td>
</tr>
<tr>
<td><code>--outdir OUTDIR</code></td>
<td>Output directory to store files created. (Default: current directory)</td>
</tr>
</tbody>
</table>

You can run the utility to decrypt encrypted NVS partition using the command below:

```python
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 for the same is provided with the utility:

```python
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 for the same is provided with the utility:

```python
python nvs_partition_gen.py generate sample_multipage_blob.csv sample.bin 0x4000 --version 2
```

#### Note:
- Minimum NVS Partition Size needed is 0x3000 bytes.
- 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.

### 2.8.6 NVS Partition Parser Utility

**Introduction**

The utility `nvs_flash/nvs_partition_tool/nvs_tool.py` loads and parses an NVS storage partition for easier debugging and data extraction. The utility also features integrity check which scans the partition for potential errors. Data blobs are encoded in `base64` format.

**Encrypted Partitions**

This utility does not support decryption. To decrypt the NVS partition, please use the NVS Partition Generator Utility which does support NVS partition encryption and decryption.
Usage

There are two output format styles available with the -f or --format option:

- `json` - All of the output is printed as a JSON.
- `text` - The output is printed as a human-readable text with different selectable output styles mentioned below.

For the `text` output format, the utility provides six different output styles with the -d or --dump option:

- `all` (default) - Prints all entries with metadata.
- `written` - Prints only written entries with metadata.
- `minimal` - Prints written namespace: key = value pairs.
- `namespaces` - Prints all written namespaces.
- `blobs` - Prints all blobs and strings (reconstructs them if they are chunked).
- `storage_info` - Prints entry states count for every page.

Note: There is also a `none` option which will not print anything. This can be used with the integrity check option if the NVS partition contents are irrelevant.

The utility also provides an integrity check feature via the -i or --integrity-check option (available only with the `text` format as it would invalidate the `json` output). This feature scans through the entire partition and prints potential errors. It can be used with the -d `none` option which will print only the potential errors.

2.8.7 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 (esp_driver_sdmmc/include/driver/sdmmc_host.h and driver/sdspi/include/driver/sdspi_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:

- SD SPI host, see SD SPI Host API

Pin Configurations

.. only:: SOC_SDMMC_USE_IOMUX and not SOC_SDMMC_USE_GPIO_MATRIX

SDMMC pins are dedicated, you don’t have to configure the pins.

.. only:: SOC_SDMMC_USE_GPIO_MATRIX and not SOC_SDMMC_USE_IOMUX

SDMMC pin signals are routed via GPIO Matrix, so you will need to configure the pins in sdmmc_slot_config_t.

.. only:: esp32p4

SDMMC have two slots:

- slot 0 pins are dedicated for UHS-I mode. This is not yet supported in the driver.
Chapter 2. API Reference

• slot 1 pins are routed via GPIO Matrix, and it’s for non UHS-I usage. You will need to configure the pins in `sdmmc_slot_config_t` to use the slot 1.

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.

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

• components/sdmmc/include/sdmmc_cmd.h
• This header file can be included with:

```c
#include "sdmmc_cmd.h"
```

• This header file is a part of the API provided by the `sdmmc` component. To declare that your component depends on `sdmmc`, add the following to your `CMakeLists.txt`:

```c
REQUIRES sdmmc
```

or

```c
PRIV_REQUIRES sdmmc
```

Functions

```c
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.

**Parameters**

• `host` - pointer to structure defining host controller
• `out_card` - pointer to structure which will receive information about the card when the function completes

**Returns**

• ESP_OK on success
• One of the error codes from SDMMC host controller

```c
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`
Chapter 2. API Reference

\texttt{esp_err_t \textbf{sdmmc_get_status} (sdmmc_card_t *card)}

Get status of SD/MMC card

**Parameters** card \(\rightarrow\) pointer to card information structure previously initialized using \texttt{sdmmc_card_init}

**Returns**
- ESP_OK on success
- One of the error codes from SDMMC host controller

\texttt{esp_err_t \textbf{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

**Parameters**
- \texttt{card} \(\rightarrow\) pointer to card information structure previously initialized using \texttt{sdmmc_card_init}
- \texttt{src} \(\rightarrow\) pointer to data buffer to read data from; data size must be equal to \texttt{sector_count} \times \texttt{card->csd.sector_size}
- \texttt{start_sector} \(\rightarrow\) sector where to start writing
- \texttt{sector_count} \(\rightarrow\) number of sectors to write

**Returns**
- ESP_OK on success or \texttt{sector_count} equal to 0
- One of the error codes from SDMMC host controller

\texttt{esp_err_t \textbf{sdmmc_read_sectors} (sdmmc_card_t *card, void *dst, size_t start_sector, size_t sector_count)}

Read given number of sectors from the SD/MMC card

**Parameters**
- \texttt{card} \(\rightarrow\) pointer to card information structure previously initialized using \texttt{sdmmc_card_init}
- \texttt{dst} \(\rightarrow\) pointer to data buffer to write into; buffer size must be at least \texttt{sector_count} \times \texttt{card->csd.sector_size}
- \texttt{start_sector} \(\rightarrow\) sector where to start reading
- \texttt{sector_count} \(\rightarrow\) number of sectors to read

**Returns**
- ESP_OK on success or \texttt{sector_count} equal to 0
- One of the error codes from SDMMC host controller

\texttt{esp_err_t \textbf{sdmmc_erase_sectors} (sdmmc_card_t *card, size_t start_sector, size_t sector_count, sdmmc_erase_arg_t arg)}

Erase given number of sectors from the SD/MMC card

**Note:** When \texttt{sdmmc_erase_sectors} used with cards in SDSPI mode, it was observed that card requires re-init after erase operation.

**Parameters**
- \texttt{card} \(\rightarrow\) pointer to card information structure previously initialized using \texttt{sdmmc_card_init}
- \texttt{start_sector} \(\rightarrow\) sector where to start erase
- \texttt{sector_count} \(\rightarrow\) number of sectors to erase
- \texttt{arg} \(\rightarrow\) erase command (CMD38) argument

**Returns**
- ESP_OK on success or \texttt{sector_count} equal to 0
- One of the error codes from SDMMC host controller

\texttt{esp_err_t \textbf{sdmmc_can_discard} (sdmmc_card_t *card)}

Check if SD/MMC card supports discard

**Parameters** card \(\rightarrow\) pointer to card information structure previously initialized using \texttt{sdmmc_card_init}

**Returns**
- ESP_OK if supported by the card/device
Chapter 2. API Reference

- ESP_FAIL if not supported by the card/device

```c
esp_err_t sdmmc_can_trim (sdmmc_card_t *card)
```
Check if SD/MMC card supports trim

Parameters
- `card` — pointer to card information structure previously initialized using `sdmmc_card_init`

Returns
- ESP_OK if supported by the card/device
- ESP_FAIL if not supported by the card/device

```c
esp_err_t sdmmc_mmc_can_sanitize (sdmmc_card_t *card)
```
Check if SD/MMC card supports sanitize

Parameters
- `card` — pointer to card information structure previously initialized using `sdmmc_card_init`

Returns
- ESP_OK if supported by the card/device
- ESP_FAIL if not supported by the card/device

```c
esp_err_t sdmmc_mmc_sanitize (sdmmc_card_t *card, uint32_t timeout_ms)
```
Sanitize the data that was unmapped by a Discard command

Note: Discard command has to precede sanitize operation. To discard, use MMC_DICARD_ARG with `sdmmc_erase_sectors` argument

Parameters
- `card` — pointer to card information structure previously initialized using `sdmmc_card_init`
- `timeout_ms` — timeout value in milliseconds required to sanitize the selected range of sectors.

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

```c
esp_err_t sdmmc_full_erase (sdmmc_card_t *card)
```
Erase complete SD/MMC card

Parameters
- `card` — pointer to card information structure previously initialized using `sdmmc_card_init`

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

```c
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)

Parameters
- `card` — pointer to card information structure previously initialized using `sdmmc_card_init`
- `function` — IO function number
- `reg` — byte address within IO function
- `out_byte` — [out] output, receives the value read from the card

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller

```c
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)

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` — input, value written to the card
- `out_byte` — [out] output, receives the value read from the card

Returns
- ESP_OK on success
- One of the error codes from SDMMC host controller
• **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_byte** - [out] if not NULL, receives new byte value read from the card (read-after-write).

**Returns**

- ESP_OK on success
- One of the error codes from SDMMC host controller

```c
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`.

**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

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size exceeds 512 bytes
- One of the error codes from SDMMC host controller

```c
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`.

**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

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_SIZE if size exceeds 512 bytes
- One of the error codes from SDMMC host controller

```c
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`.

**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.

**Returns**

- 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

```c
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`.

**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.

**Returns**
- 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

```c
esp_err_t sdmmc_io_enable_int(sdmmc_card_t* card)
```

Enable SDIO interrupt in the SDMMC host

**Parameters**
- **card** - pointer to card information structure previously initialized using `sdmmc_card_init`

**Returns**
- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if the host controller does not support IO interrupts

```c
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.

**Parameters**
- **card** - pointer to card information structure previously initialized using `sdmmc_card_init`
- **timeout_ticks** - time to wait for the interrupt, in RTOS ticks

**Returns**
- 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

```c
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 an SDIO card.

You may provide a buffer not sufficient to store all the CIS data. In this case, this function stores as much data into your buffer as possible. Also, this function will try to get and return the size required for you.

**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.

**Returns**
- 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

`esp_err_t sdmmc_io_print_cis_info(uint8_t* buffer, size_t buffer_size, FILE *fp)`

Parse and print the CIS information of an 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.

**Parameters**
- `buffer` - Buffer to parse
- `buffer_size` - Size of the buffer.
- `fp` - File pointer to print to, set to NULL to print to stdout.

**Returns**
- 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.

**Header File**

- `components/esp_driver_sdmmc/include/driver/sdmmc_types.h`

This header file can be included with:

```c
#include "driver/sdmmc_types.h"
```

- This header file is a part of the API provided by the `esp_driver_sdmmc` component. To declare that your component depends on `esp_driver_sdmmc`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_driver_sdmmc
```

or

```cmake
PRIV_REQUIRES esp_driver_sdmmc
```

### 2.8.8 Partitions API

**Overview**

The `esp_partition` component has higher-level API functions which work with partitions defined in the **Partition Tables**. These APIs are based on lower level API provided by **SPI Flash API**.

**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 can be found in **Partition Tables**.

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()` is 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 `esp_flash_read()`, `esp_flash_write()`, `esp_flash_erase_region()`, but operate within partition boundaries.

**See Also**

• *Partition Tables*
• *Over The Air Updates (OTA)* provides high-level API for updating applications stored in flash.
• *Non-Volatile Storage Library* provides a structured API for storing small pieces of data in SPI flash.

**API Reference - Partition Table**

**Header File**

• `components/esp_partition/include/esp_partition.h`

This header file can be included with:

```
#include "esp_partition.h"
```

This header file is a part of the API provided by the `esp_partition` component. To declare that your component depends on `esp_partition`, add the following to your CMakeLists.txt:

```
REQUIRES esp_partition
```

or

```
PRIV_REQUIRES esp_partition
```

**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.

**Parameters**

• `type` – Partition type, one of esp_partition_type_t values or an 8-bit unsigned integer. To find all partitions, no matter the type, use ESP_PARTITION_TYPE_ANY, and set subtype argument to ESP_PARTITION_SUBTYPE_ANY.

• `subtype` – Partition subtype, one of esp_partition_subtype_t values or an 8-bit unsigned integer. 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.

**Returns** 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.

`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.

**Parameters**

• `type` – Partition type, one of esp_partition_type_t values or an 8-bit unsigned integer. To find all partitions, no matter the type, use ESP_PARTITION_TYPE_ANY, and set subtype argument to ESP_PARTITION_SUBTYPE_ANY.
• **subtype** – Partition subtype, one of esp_partition_subtype_t values or an 8-bit unsigned integer. 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.

Returns: pointer to esp_partition_t structure, or NULL if no partition is found. This pointer is valid for the lifetime of the application.

```c
const esp_partition_t *esp_partition_get (esp_partition_iterator_t iterator)
```

Get esp_partition_t structure for given partition.

Parameters: iterator – Iterator obtained using esp_partition_find. Must be non-NULL.

Returns: pointer to esp_partition_t structure. This pointer is valid for the lifetime of the application.

```c
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.

Parameters: iterator – Iterator obtained using esp_partition_find. Must be non-NULL.

Returns: NULL if no partition was found, valid esp_partition_iterator_t otherwise.

```c
void esp_partition_iterator_release (esp_partition_iterator_t iterator)
```

Release partition iterator.

Parameters: iterator – Iterator obtained using esp_partition_find. The iterator is allowed to be NULL, so it is not necessary to check its value before calling this function.

```c
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.

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.

Returns:
• 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.

```c
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.

Partitions marked with an encryption flag will automatically be be read and decrypted via a cache mapping.

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.

Returns: 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.

```c
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 esp_flash_write_encrypted() function. If writing to an encrypted partition, all write offsets and lengths must be multiples of 16 bytes. See the esp_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.

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.

Returns 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; ESP_ERR_NOT_ALLOWED, if partition is read-only; or one of error codes from lower-level flash driver.

```c
esp_err_t esp_partition_read_raw(const esp_partition_t *partition, size_t src_offset, void *dst, size_t size)
```

Read data from the partition without any transformation/decryption.

Note: This function is essentially the same as `esp_partition_read()` above. It just never decrypts data but returns it as is.

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.

Returns 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.

```c
esp_err_t esp_partition_write_raw(const esp_partition_t *partition, size_t dst_offset, const void *src, size_t size)
```

Write data to the partition without any transformation/encryption.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using esp_partition_erase_range function.

Note: This function is essentially the same as `esp_partition_write()` above. It just never encrypts data but writes it as is.
Chapter 2. API Reference

Note: Prior to writing to flash memory, make sure it has been erased with esp_partition_erase_range call.

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.

Returns 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; ESP_ERR_NOT_ALLOWED, if partition is read-only; or one of the error codes from lower-level flash driver.

\[\text{esp_err_t esp_partition_erase_range}(\text{const esp_partition_t *partition, size_t offset, size_t size})\]
Erase part of the partition.

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 partition->erase_size.
• size – Size of the range which should be erased, in bytes. Must be divisible by partition->erase_size.

Returns 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; ESP_ERR_NOT_ALLOWED, if partition is read-only; or one of error codes from lower-level flash driver.

\[\text{esp_err_t esp_partition_mmap}(\text{const esp_partition_t *partition, size_t offset, size_t size, esp_partition_mmap_memory_t memory, const void **out_ptr, esp_partition_mmap_handle_t *out_handle})\]
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 esp_partition_munmap function.

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 esp_partition_munmap call

Returns ESP_OK, if successful

\[\text{void esp_partition_munmap}(\text{esp_partition_mmap_handle_t handle})\]
Release region previously obtained using esp_partition_mmap.

Note: Calling this function will not necessarily unmap memory region. Region will only be unmapped when there are no other handles which reference this region. In case of partially overlapping regions it is possible
that memory will be unmapped partially.

Parameters handle – Handle obtained from spi_flash_mmap

```c
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.

Parameters

- **partition** – [in] Pointer to info for partition containing app or data. (fields: address, size and type, are required to be filled).
- **sha_256** – [out] Returned SHA-256 digest for a given partition.

Returns

- 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.

```c
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.

Parameters

- **partition_1** – [in] Pointer to info for partition 1 containing app or data. (fields: address, size and type, are required to be filled).
- **partition_2** – [in] Pointer to info for partition 2 containing app or data. (fields: address, size and type, are required to be filled).

Returns

- True: In case of the two firmware is equal.
- False: Otherwise

```c
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.

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_*), or an integer. 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_*), or an integer.
- **out_partition** – [out] Output, if non-NULL, receives the pointer to the resulting `esp_partition_t` structure

Returns

- ESP_OK on success
- ESP_ERR_NO_MEM if memory allocation has failed
Chapter 2. API Reference

- 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

`esp_err_t esp_partition_deregister_external (const esp_partition_t *partition)`

Deregister the partition previously registered using `esp_partition_register_external`.

**Parameters**
- `partition` - pointer to the partition structure obtained from `esp_partition_register_external`.

**Returns**
- 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.

**Structures**

```c
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**

```c
esp_flash_t *flash_chip
```

SPI flash chip on which the partition resides

```c
esp_partition_type_t type
```

Partition type (app/data)

```c
esp_partition_subtype_t subtype
```

Partition subtype

```c
uint32_t address
```

Starting address of the partition in flash

```c
uint32_t size
```

Size of the partition, in bytes

```c
uint32_t erase_size
```

Size the erase operation should be aligned to

```c
char label[17]
```

Partition label, zero-terminated ASCII string

```c
bool encrypted
```

Flag is set to true if partition is encrypted

```c
bool readonly
```

Flag is set to true if partition is read-only
Chapter 2. API Reference

Macros

`ESP_PARTITION_SUBTYPE_OTA (i)`
Convenience macro to get `esp_partition_subtype_t` value for the i-th OTA partition.

Type Definitions

`typedef uint32_t esp_partition_mmap_handle_t`
Opaque handle for memory region obtained from `esp_partition_mmap`.

`typedef struct esp_partition_iterator_opaque_ * esp_partition_iterator_t`
Opaque partition iterator type.

Enumerations

`enum esp_partition_mmap_memory_t`
Enumeration which specifies memory space requested in an mmap call.

Values:

enumerator `ESP_PARTITION_MMAP_DATA`
map to data memory (Vaddr0), allows byte-aligned access, 4 MB total

enumerator `ESP_PARTITION_MMAP_INST`
map to instruction memory (Vaddr1-3), allows only 4-byte-aligned access, 11 MB total

`enum esp_partition_type_t`
Partition type.

Note: Partition types with integer value 0x00-0x3F are reserved for partition types defined by ESP-IDF. Any other integer value 0x40-0xFE can be used by individual applications, without restriction.

Values:

enumerator `ESP_PARTITION_TYPE_APP`
Application partition type.

enumerator `ESP_PARTITION_TYPE_DATA`
Data partition type.

enumerator `ESP_PARTITION_TYPE_ANY`
Used to search for partitions with any type.

`enum esp_partition_subtype_t`
Partition subtype.

Application-defined partition types (0x40-0xFE) can set any numeric subtype value.

Note: These ESP-IDF-defined partition subtypes apply to partitions of type `ESP_PARTITION_TYPE_APP` and `ESP_PARTITION_TYPE_DATA`.

Values:
enumerator ESP_PARTITION_SUBTYPE_APP_FACTORY
   Factory application partition.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_MIN
   Base for OTA partition subtypes.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_0
   OTA partition 0.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_1
   OTA partition 1.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_2
   OTA partition 2.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_3
   OTA partition 3.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_4
   OTA partition 4.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_5
   OTA partition 5.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_6
   OTA partition 6.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_7
   OTA partition 7.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_8
   OTA partition 8.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_9
   OTA partition 9.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_10
   OTA partition 10.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_11
   OTA partition 11.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_12
   OTA partition 12.

enumerator ESP_PARTITION_SUBTYPE_APP_OTA_13
   OTA partition 13.
enumerator `ESP_PARTITION_SUBTYPE_APP_OTA_14`
OTA partition 14.

enumerator `ESP_PARTITION_SUBTYPE_APP_OTA_15`
OTA partition 15.

enumerator `ESP_PARTITION_SUBTYPE_APP_OTA_MAX`
Max subtype of OTA partition.

enumerator `ESP_PARTITION_SUBTYPE_APP_TEST`
Test application partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_OTA`
OTA selection partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_PHY`
PHY init data partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_NVS`
NVS partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_COREDUMP`
COREDUMP partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_NVS_KEYS`
Partition for NVS keys.

enumerator `ESP_PARTITION_SUBTYPE_DATA_EFUSE_EM`
Partition for emulate eFuse bits.

enumerator `ESP_PARTITION_SUBTYPE_DATA_UNDEFINED`
Undefined (or unspecified) data partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_ESPHTTPD`
ESPHTTPD partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_FAT`
FAT partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_SPIFFS`
SPIFFS partition.

enumerator `ESP_PARTITION_SUBTYPE_DATA_LITTLEFS`
LITTLEFS partition.

enumerator `ESP_PARTITION_SUBTYPE_ANY`
Used to search for partitions with any subtype.
2.8.9 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.
- SPIFFS is able to reliably utilize only around 75% of assigned partition space.
- When the filesystem is running out of space, the garbage collector is trying to find free space by scanning the filesystem multiple times, which can take up to several seconds per write function call, depending on required space. This is caused by the SPIFFS design and the issue has been reported multiple times (e.g., here) and in the official SPIFFS github repository. The issue can be partially mitigated by the SPIFFS configuration.
- Deleting a file does not always remove the whole file, which leaves unusable sections throughout the filesystem.
- When the chip experiences a power loss during a file system operation it could result in SPIFFS corruption. However the file system still might be recovered via esp_spiffs_check function. More details in the official SPIFFS FAQ.

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:

```
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.

`spiffs_create_partition_image` must be called from one of the component CMakeLists.txt files.

 Optionally, users can opt to have the image automatically flashed together with the app binaries, partition tables, etc. on `idf.py flash` by specifying `FLASH_IN_PROJECT`. For example:

```python
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:

```cpp
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:

```bash
mkspiffs -c [src_folder] -b 4096 -p 256 -s 0x100000 spiffs.bin
```

To flash the image onto ESP32-S2 at offset 0x110000, run:

```bash
python esptool.py --chip esp32s2 --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 `mkspiffs` 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.
Chapter 2. API Reference

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

- components/spiffs/include/esp_spiffs.h
  - This header file can be included with:
    ```
    #include "esp_spiffs.h"
    ```
  - This header file is a part of the API provided by the spiffs component. To declare that your component depends on spiffs, add the following to your CMakeLists.txt:
    ```
    REQUIRE spiffs
    ```
    or
    ```
    PRIV_REQUIRES spiffs
    ```

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.
  - **Parameters** conf — Pointer to `esp_vfs_spiffs_conf_t` configuration structure
  - **Returns**
    - 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

- `esp_err_t esp_vfs_spiffs_unregister(const char* partition_label)`
  - Unregister and unmount SPIFFS from VFS
  - **Parameters** partition_label — Same label as passed to `esp_vfs_spiffs_register`.
  - **Returns**
    - ESP_OK if successful
    - ESP_ERR_INVALID_STATE already unregistered

- `bool esp_spiffs_mounted(const char* partition_label)`
  - Check if SPIFFS is mounted
  - **Parameters** partition_label — Optional, label of the partition to check. If not specified, first partition with subtype=spiffs is used.
  - **Returns**
    - true if mounted
    - false if not mounted

- `esp_err_t esp_spiffs_format(const char* partition_label)`
  - Format the SPIFFS partition
  - **Parameters** partition_label — Same label as passed to `esp_vfs_spiffs_register`. 
Returns

- ESP_OK if successful
- ESP_FAIL on error

`esp_err_t esp_spiffs_info(const char *partition_label, size_t *total_bytes, size_t *used_bytes)`

Get information for SPIFFS

Parameters

- `partition_label` - Same label as passed to esp_vfs_spiffs_register
- `total_bytes` - [out] Size of the file system
- `used_bytes` - [out] Current used bytes in the file system

Returns

- ESP_OK if success
- ESP_ERR_INVALID_STATE if not mounted

`esp_err_t esp_spiffs_check(const char *partition_label)`

Check integrity of SPIFFS

Parameters `partition_label` - Same label as passed to esp_vfs_spiffs_register

Returns

- ESP_OK if successful
- ESP_ERR_INVALID_STATE if not mounted
- ESP_FAIL on error

`esp_err_t esp_spiffs_gc(const char *partition_label, size_t size_to_gc)`

Perform garbage collection in SPIFFS partition.

Call this function to run GC and ensure that at least the given amount of space is available in the partition. This function will fail with ESP_ERR_NOT_FINISHED if it is not possible to reclaim the requested space (that is, not enough free or deleted pages in the filesystem). This function will also fail if it fails to reclaim the requested space after CONFIG_SPIFFS_GC_MAX_RUNS number of GC iterations. On one GC iteration, SPIFFS will erase one logical block (4kB). Therefore the value of CONFIG_SPIFFS_GC_MAX_RUNS should be set at least to the maximum expected size_to_gc, divided by 4096. For example, if the application expects to make room for a 1MB file and calls esp_spiffs_gc(label, 1024 * 1024), CONFIG_SPIFFS_GC_MAX_RUNS should be set to at least 256. On the other hand, increasing CONFIG_SPIFFS_GC_MAX_RUNS value increases the maximum amount of time for which any SPIFFS GC or write operation may potentially block.

Parameters

- `partition_label` - Label of the partition to be garbage-collected. The partition must be already mounted.
- `size_to_gc` - The number of bytes that the GC process should attempt to make available.

Returns

- ESP_OK on success
- ESP_ERR_NOT_FINISHED if GC fails to reclaim the size given by size_to_gc
- ESP_ERR_INVALID_STATE if the partition is not mounted
- ESP_FAIL on all other errors

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.

2.8.10 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 printf, 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"). Then the VFS component calls 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:

```c
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):

```c
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
```

(continues on next page)
Case 2: API functions are declared with an extra context pointer (the FS driver supports multiple instances):

```c
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 de-initialization of the environment for checking events.
6. The `select()` call ends and returns the appropriate results.

### Non-WebSocket 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:

```c
// 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 (i.e., should not crash but return an error instead).
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:

```c
// 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 is 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/esp32xx/vfs_lwip.c for a reference socket driver implementation using LWIP.

---

**Note:** If you use select() for socket file descriptors only then you can disable the CONFIG_VFS_SUPPORT_SELECT option to reduce the code size and improve performance. You should not 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 sys/select.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 sends characters to the UART transmit FIFO. Reading from stdin retrieves 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:

```c
fprintf(stderr, "%d\n");
```

The __getreent() function returns a per-task pointer to struct _reent in newlib libc. 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 closes 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.

eventfd()
eventfd() call is a powerful tool to notify a select() based loop of custom events. The eventfd() implementation in ESP-IDF is generally the same as described in man(2) eventfd except for:

• esp_vfs_eventfd_register() has to be called before calling eventfd()
• Options EFD_CLOEXEC, EFD_NONBLOCK and EFD_SEMAPHORE are not supported in flags.
• Option EFD_SUPPORT_ISR has been added in flags. This flag is required to read and write the eventfd in an interrupt handler.

Note that creating an eventfd with EFD_SUPPORT_ISR will cause interrupts to be temporarily disabled when reading, writing the file and during the beginning and the ending of the select() when this file is set.

API Reference

Header File

• components/vfs/include/esp_vfs.h
• This header file can be included with:

```
#include "esp_vfs.h"
```

• This header file is a part of the API provided by the vfs component. To declare that your component depends on vfs, add the following to your CMakeLists.txt:

```
REQUIRES vfs
```

or

```
PRIV_REQUIRES vfs
```

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.

Parameters
- **base_path** – file path prefix associated with the filesystem. Must be a zero-terminated C string, may be empty. If not empty, must be 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”. In the special case of an empty base_path, a “fallback” VFS is registered. Such VFS will handle paths which are not matched by any other registered VFS.
- **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.

Returns
ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered.

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.

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).

Returns
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.

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.

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.

Returns
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.

esp_err_t esp_vfs_unregister (const char *base_path)

Unregister a virtual filesystem for given path prefix

Parameters **base_path** – file prefix previously used in esp_vfs_register call

Returns
ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for given prefix hasn’t been registered.

esp_err_t esp_vfs_unregister_with_id (esp_vfs_id_t vfs_id)

Unregister a virtual filesystem with the given index

Parameters **vfs_id** – The VFS ID returned by esp_vfs_register_with_id

Returns
ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for the given index hasn’t been registered.

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.

Parameters

• **vfs_id** - VFS identifier returned by `esp_vfs_register_with_id`

• **fd** - The registered file descriptor will be written to this address.

**Returns**  
- 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.

```c
esp_err_t esp_vfs_register_fd_with_local_fd(esp_vfs_id_t vfs_id, int local_fd, bool permanent, int *fd)
```

Special function for registering another file descriptor with given `local_fd` for a VFS registered by `esp_vfs_register_with_id`.

**Parameters**

• **vfs_id** - VFS identifier returned by `esp_vfs_register_with_id`.

• **local_fd** - The fd in the local vfs. Passing -1 will set the local fd as the (*fd) value.

• **permanent** – Whether the fd should be treated as permanent (not removed after close())

• **fd** - The registered file descriptor will be written to this address.

**Returns**  
- 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.

```c
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`.

**Parameters**

• **vfs_id** - VFS identifier returned by `esp_vfs_register_with_id`.

• **fd** - File descriptor which should be unregistered.

**Returns**  
- ESP_OK if the registration is successful, ESP_ERR_INVALID_ARG if the arguments are incorrect.

```c
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.

**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.

**Returns**  
- The number of descriptors set in the descriptor sets, or -1 when an error (specified by `errno`) have occurred.

```c
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

```c
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`.
Chapter 2. API Reference

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

ssize_t esp_vfs_pread (int fd, void *dst, size_t size, off_t offset)

Implements the VFS layer of POSIX pread()

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

Returns

A positive return value indicates the number of bytes read. -1 is return on failure and errno is set accordingly.

ssize_t esp_vfs_pwrite (int fd, const void *src, size_t size, off_t offset)

Implements the VFS layer of POSIX pwrite()

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

Returns

A positive return value indicates the number of bytes written. -1 is return on failure and errno is set accordingly.

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 and/or ESP_VFS_FLAG_READONLY_FS or ESP_VFS_FLAG_DEFAULT

ssize_t (*write_p)(void *p, int fd, const void *data, size_t size)
Write with context pointer

ssize_t (*write)(int fd, const void *data, size_t size)
Write without context pointer

off_t (*lseek_p)(void *p, int fd, off_t size, int mode)
Seek with context pointer

off_t (*lseek)(int fd, off_t size, int mode)
Seek without context pointer

ssize_t (*read_p)(void *ctx, int fd, void *dst, size_t size)
Read with context pointer

ssize_t (*read)(int fd, void *dst, size_t size)
Read without context pointer

ssize_t (*pread_p)(void *ctx, int fd, void *dst, size_t size, off_t offset)
pread with context pointer

ssize_t (*pread)(int fd, void *dst, size_t size, off_t offset)
pread without context pointer

ssize_t (*pwrite_p)(void *ctx, int fd, const void *src, size_t size, off_t offset)
pwrite with context pointer

ssize_t (*pwrite)(int fd, const void *src, size_t size, off_t offset)
pwrite without context pointer

int (*open_p)(void *ctx, const char *path, int flags, int mode)
open with context pointer

int (*open)(const char *path, int flags, int mode)
open without context pointer

int (*close_p)(void *ctx, int fd)
close with context pointer

int (*close)(int fd)
close without context pointer

int (*fstat_p)(void *ctx, int fd, struct stat *st)
fstat with context pointer
int (*fstat)(int fd, struct stat *st)
   fstat without context pointer

int (*stat_p)(void *ctx, const char *path, struct stat *st)
   stat with context pointer

int (*stat)(const char *path, struct stat *st)
   stat without context pointer

int (*link_p)(void *ctx, const char *n1, const char *n2)
   link with context pointer

int (*link)(const char *n1, const char *n2)
   link without context pointer

int (*unlink_p)(void *ctx, const char *path)
   unlink with context pointer

int (*unlink)(const char *path)
   unlink without context pointer

int (*rename_p)(void *ctx, const char *src, const char *dst)
   rename with context pointer

int (*rename)(const char *src, const char *dst)
   rename without context pointer

DIR *(opendir_p)(void *ctx, const char *name)
   opendir with context pointer

DIR *(opendir)(const char *name)
   opendir without context pointer

struct dirent *(readdir_p)(void *ctx, DIR *pdir)
   readdir with context pointer

struct dirent *(readdir)(DIR *pdir)
   readdir without context pointer

int (*readdir_r_p)(void *ctx, DIR *pdir, struct dirent *entry, struct dirent **out_dirent)
   readdir_r with context pointer

int (*readdir_r)(DIR *pdir, struct dirent *entry, struct dirent **out_dirent)
   readdir_r without context pointer

long (*telldir_p)(void *ctx, DIR *pdir)
   telldir with context pointer
long (*telldir)(DIR *pdir)
    telldir without context pointer

void (*seekdir_p)(void *ctx, DIR *pdir, long offset)
    seekdir with context pointer

void (*seekdir)(DIR *pdir, long offset)
    seekdir without context pointer

int (*closedir_p)(void *ctx, DIR *pdir)
    closedir with context pointer

int (*closedir)(DIR *pdir)
    closedir without context pointer

int (*mkdir_p)(void *ctx, const char *name, mode_t mode)
    mkdir with context pointer

int (*mkdir)(const char *name, mode_t mode)
    mkdir without context pointer

int (*rmdir_p)(void *ctx, const char *name)
    rmdir with context pointer

int (*rmdir)(const char *name)
    rmdir without context pointer

int (*fcntl_p)(void *ctx, int fd, int cmd, int arg)
    fcntl with context pointer

int (*fcntl)(int fd, int cmd, int arg)
    fcntl without context pointer

int (*ioctl_p)(void *ctx, int fd, int cmd, va_list args)
    ioctl with context pointer

int (*ioctl)(int fd, int cmd, va_list args)
    ioctl without context pointer

int (*fsync_p)(void *ctx, int fd)
    fsync with context pointer

int (*fsync)(int fd)
    fsync without context pointer

int (*access_p)(void *ctx, const char *path, int amode)
    access with context pointer
int (*access)(const char *path, int amode)
    access without context pointer

int (*truncate_p)(void *ctx, const char *path, off_t length)
    truncate with context pointer

int (*truncate)(const char *path, off_t length)
    truncate without context pointer

int (*ftruncate_p)(void *ctx, int fd, off_t length)
    ftruncate with context pointer

int (*ftruncate)(int fd, off_t length)
    ftruncate without context pointer

int (*utime_p)(void *ctx, const char *path, const struct utimbuf *times)
    utime with context pointer

int (*utime)(const char *path, const struct utimbuf *times)
    utime without context pointer

int (*tcsetattr_p)(void *ctx, int fd, int optional_actions, const struct termios *p)
    tcsetattr with context pointer

int (*tcsetattr)(int fd, int optional_actions, const struct termios *p)
    tcsetattr without context pointer

int (*tcgetattr_p)(void *ctx, int fd, struct termios *p)
    tcgetattr with context pointer

int (*tcgetattr)(int fd, struct termios *p)
    tcgetattr without context pointer

int (*tcdrain_p)(void *ctx, int fd)
    tcdrain with context pointer

int (*tcdrain)(int fd)
    tcdrain without context pointer

int (*tcflush_p)(void *ctx, int fd, int select)
    tcflush with context pointer

int (*tcflush)(int fd, int select)
    tcflush without context pointer

int (*tcflow_p)(void *ctx, int fd, int action)
    tcflow with context pointer
int (*tcflow)(int fd, int action)
    tcflow without context pointer

pid_t (*tcgetsid_p)(void *ctx, int fd)
    tcgetsid with context pointer

pid_t (*tcgetsid)(int fd)
    tcgetsid without context pointer

int (*tcsendbreak_p)(void *ctx, int fd, int duration)
    tcsendbreak with context pointer

int (*tcsendbreak)(int fd, int duration)
    tcsendbreak without context pointer

esp_err_t (*start_select)(int nfds, 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 nfds, 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.
Chapter 2. API Reference

**ESP_VFS_FLAG_CONTEXT_PTR**
Flag which indicates that FS needs extra context pointer in syscalls.

**ESP_VFS_FLAG_READONLY_FS**
Flag which indicates that FS is located on read-only partition.

Type Definitions

typedef int esp_vfs_id_t

Header File

- `components/vfs/include/esp_vfs_dev.h`
- This header file can be included with:
  ```c
  #include "esp_vfs_dev.h"
  ```
- This header file is a part of the API provided by the vfs component. To declare that your component depends on vfs, add the following to your CMakeLists.txt:
  ```cmake
  REQUIRE vfs
  ```
  or
  ```cmake
  PRIV_REQUIRE vfs
  ```

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:
Chapter 2. API Reference

- 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**
- **uart_num** – the UART number
- **mode** – line endings to send to UART

**Returns**
0 if succeeded, or -1 when an error (specified by errno) have occurred.

```
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

**Parameters**
- **uart_num** – UART peripheral number
- **mode** – line endings to send to UART

**Returns**
0 if succeeded, or -1 when an error (specified by errno) have occurred.

```
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

**Parameters**
- ** uart_num** – the UART number
- ** mode** – line endings to send to UART

**Returns**
0 if succeeded, or -1 when an error (specified by errno) have occurred.

```
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

void esp_vfs_usb_serial_jtag_use_driver (void)
set VFS to use USB-SERIAL-JTAG driver for reading and writing

Note: application must configure USB-SERIAL-JTAG driver before calling these functions. With these functions, read and write are blocking and interrupt-driven.

void esp_vfs_usb_serial_jtag_use_nonblocking (void)
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.

Header File

- components/vfs/include/esp_vfs_eventfd.h
- This header file can be included with:

```
#include "esp_vfs_eventfd.h"
```

- This header file is a part of the API provided by the vfs component. To declare that your component depends on vfs, add the following to your CMakeLists.txt:

```
REQUIRES vfs
```
or

```
PRIV_REQUIRES vfs
```

Functions

```
esp_err_t esp_vfs_eventfd_register (const esp_vfs_eventfd_config_t *config)
```
Registers the event vfs.

Returns ESP_OK if successful, ESP_ERR_NO_MEM if too many VFSes are registered.

```
esp_err_t esp_vfs_eventfd_unregister (void)
```
Unregisters the event vfs.

Returns ESP_OK if successful, ESP_ERR_INVALID_STATE if VFS for given prefix hasn’t been registered

```
int eventfd (unsigned int initval, int flags)
```

Structures

```
struct esp_vfs_eventfd_config_t
```
Eventfd vfs initialization settings.
Chapter 2. API Reference

Public Members

size_t max_fds
The maximum number of eventfd supported

Macros

EFD_SUPPORT_ISR
ESP_VFS_EVENTD_CONFIG_DEFAULT ()

2.8.11 Wear Levelling API

Overview

Most of flash memory and especially SPI flash that is used in ESP32-S2 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
- **wlUnmount** - 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 Support
- Partition Tables

Application Example

An example that 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 storage/wear_levelling/README.md for more information.

High-level API Reference

Header Files

- fatfs/vfs/esp_vfs_fat.h

High-level wear levelling functions esp_vfsFatSpiFlashMount_RW_WL(), esp_vfsFatSpiFlashUnmount_RW_WL() and struct esp_vfsFatMount_config_t are described in FAT Filesystem Support.

Mid-level API Reference

Header File

- components/wear_levelling/include/wear_levelling.h

This header file can be included with:

```c
#include "wear_levelling.h"
```

This header file is a part of the API provided by the wear_levelling component. To declare that your component depends on wear_levelling, add the following to your CMakeLists.txt:

```cmake
REQUIRES wear_levelling
```

or

```cmake
PRIV_REQUIRES wear_levelling
```

Functions

```c
esp_err_t esp_vfsFatSpiFlashMount_RW_WL(const esp_partition_t *partition, wl_handle_t *out_handle)
```

Mount WL for defined partition.

Parameters

- `partition` - that will be used for access
- `out_handle` - handle of the WL instance

Returns

- ESP_OK, if the WL allocation is successful;
- ESP_ERR_INVALID_ARG, if the arguments for WL configuration are not valid;
- ESP_ERR_NO_MEM, if the WL allocation fails because of insufficient memory;
**esp_err_t** *wl_unmount*( *wl_handle_t* handle)

Unmount WL for defined partition.

**Parameters**
- **handle** – WL partition handle

**Returns**
- ESP_OK, if the operation is successful;
- or one of error codes from lower-level flash driver.

**esp_err_t** *wl_erase_range*( *wl_handle_t* handle, size_t start_addr, size_t size)

Erase part of the WL storage.

**Parameters**
- **handle** – WL handle that are related to the partition
- **start_addr** – Address from 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 the result of function *wl_sector_size*( … ).

**Returns**
- ESP_OK, if the given 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.

**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.

**Parameters**
- **handle** – WL handle corresponding to the WL 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.

**Returns**
- 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.

**esp_err_t** *wl_read*( *wl_handle_t* handle, size_t src_addr, void* dest, size_t size)

Read data from the WL storage.

**Parameters**
- **handle** – WL module instance that was initialized before
- **dest** – Pointer to the buffer where data should be stored. The Pointer must be non-NULL and the 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.

**Returns**
- 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.
size_t \text{wl\_size}(\text{wl\_handle\_t handle})

Get the actual flash size in use for the WL storage partition.

**Parameters**
- handle: WL module handle that was initialized before

**Returns**
- usable size, in bytes

size_t \text{wl\_sector\_size}(\text{wl\_handle\_t handle})

Get sector size of the WL instance.

**Parameters**
- handle: WL module handle that was initialized before

**Returns**
- sector size, in bytes

### Macros

- **WL_INVALID_HANDLE**

### Type Definitions

typedef int32_t \text{wl\_handle\_t}

wear levelling handle

**Code examples for this API section are provided in the storage directory of ESP-IDF examples.**

## 2.9 System API

### 2.9.1 App Image Format

**Application Image Structures**

An application image consists of the following:

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-S2’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:

```bash
esptool.py --chip esp32s2 image_info build/app.bin
```
For more details on the type of memory segments and their address ranges, see ESP32-S2 Technical Reference Manual > System and Memory > Internal Memory [PDF].

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 the SHA256 hash is calculated on the range from the first byte and up to this field. The length of this field is 32 bytes.

5. If the option CONFIG_SECURE_SIGNED_APPS_SCHEME is set to ECDSA then the application image will have an additional 68 bytes for an ECDSA signature, which includes:
   - version word (4 bytes)
Chapter 2. API Reference

- signature data (64 bytes)

6. If the option `CONFIG_SECURE_SIGNED_APPS_SCHEME` is set to RSA or ECDSA (V2) then the application image will have an additional signature sector of 4 KB in size. For more details on the format of this signature sector, please refer to *Signature Block Format*.

Application Description

The DROM segment of the application binary starts with the `esp_app_desc_t` structure which carries specific fields describing the application:

- magic_word: the magic word for the `esp_app_desc_t` structure
- secure_version: see *Anti-rollback*
- version: see *App version*
- project_name: filled from `PROJECT_NAME`
- time and date: compile time and date
- idf_ver: version of ESP-IDF
- `app_elf_sha256`: contains SHA256 hash for the application ELF file

This structure is useful for identification of images uploaded via Over-the-Air (OTA) updates because it has a fixed offset = `sizeof(esp_image_header_t) + sizeof(esp_image_segment_header_t)` + `sizeof(esp_app_desc_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 with or not.

To obtain the `esp_app_desc_t` structure for the currently running application, use `esp_app_get_description()`.

To obtain the `esp_app_desc_t` structure for another OTA partition, use `esp_ota_get_partition_description()`.

Adding a Custom Structure to an Application

Users also have 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:

```c
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 `target_link_libraries(${COMPONENT_TARGET} "-u custom_app_desc")` into `CMakeLists.txt`.

API Reference

Header File

- `components/bootloader_support/include/esp_app_format.h`

This header file can be included with:

```c
#include "esp_app_format.h"
```

This header file is a part of the API provided by the `bootloader_support` component. To declare that your component depends on `bootloader_support`, add the following to your `CMakeLists.txt`:

```cmake
target_link_libraries(${COMPONENT_TARGET} "-u custom_app_desc")
```

---

1 The maximum length is 32 characters, including null-termination character. For example, if the length of `PROJECT_NAME` exceeds 31 characters, the excess characters will be disregarded.
REQUIRES bootloader_support

or

PRIV_REQUIRES bootloader_support

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
    flash frequency (esp_image_spi_freq_t as uint8_t)

uint8_t spi_size
    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
    Minimal chip revision supported by image After the Major and Minor revision eFuses were introduced into the chips, this field is no longer used. But for compatibility reasons, we keep this field and the data in it. Use min_chip_rev_full instead. The software interprets this as a Major version for most of the chips and as a Minor version for the ESP32-C3.

uint16_t min_chip_rev_full
    Minimal chip revision supported by image, in format: major * 100 + minor
#### uint16_t `max_chip_rev_full`
Maximal chip revision supported by image, in format: major * 100 + minor

#### uint8_t `reserved[4]`
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

#### 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.

#### Enumerations

##### enum `esp_chip_id_t`
ESP chip ID. 

*Values:*

- enumerator `ESP_CHIP_ID_ESP32` chip ID: ESP32
- enumerator `ESP_CHIP_ID_ESP32S2` chip ID: ESP32-S2
- enumerator `ESP_CHIP_ID_ESP32C3` chip ID: ESP32-C3
- enumerator `ESP_CHIP_ID_ESP32S3` chip ID: ESP32-S3
enumerator **ESP_CHIP_ID_ESP32C2**
chip ID: ESP32-C2

enumerator **ESP_CHIP_ID_ESP32C6**
chip ID: ESP32-C6

enumerator **ESP_CHIP_ID_ESP32H2**
chip ID: ESP32-H2

enumerator **ESP_CHIP_ID_ESP32P4**
chip ID: ESP32-P4

enumerator **ESP_CHIP_ID_INVALID**
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:

enumerator **ESP_IMAGE_SPI_MODE_QIO**
SPI mode QIO

enumerator **ESP_IMAGE_SPI_MODE_QOUT**
SPI mode QOUT

enumerator **ESP_IMAGE_SPI_MODE_DIO**
SPI mode DIO

enumerator **ESP_IMAGE_SPI_MODE_DOUT**
SPI mode DOUT

enumerator **ESP_IMAGE_SPI_MODE_FAST_READ**
SPI mode FAST_READ

enumerator **ESP_IMAGE_SPI_MODE_SLOW_READ**
SPI mode SLOW_READ

**enum esp_image_spi_freq_t**

SPI flash clock division factor.

Values:

enumerator **ESP_IMAGE_SPI_SPEED_DIV_2**
The SPI flash clock frequency is divided by 2 of the clock source

enumerator **ESP_IMAGE_SPI_SPEED_DIV_3**
The SPI flash clock frequency is divided by 3 of the clock source

enumerator **ESP_IMAGE_SPI_SPEED_DIV_4**
The SPI flash clock frequency is divided by 4 of the clock source
Chapter 2. API Reference

enumerator **ESP_IMAGE_SPI_SPEED_DIV_1**
   The SPI flash clock frequency equals to the clock source

enum **esp_image_flash_size_t**
   Supported SPI flash sizes.
   **Values:**

   enumerator **ESP_IMAGE_FLASH_SIZE_1MB**
      SPI flash size 1 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_2MB**
      SPI flash size 2 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_4MB**
      SPI flash size 4 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_8MB**
      SPI flash size 8 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_16MB**
      SPI flash size 16 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_32MB**
      SPI flash size 32 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_64MB**
      SPI flash size 64 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_128MB**
      SPI flash size 128 MB

   enumerator **ESP_IMAGE_FLASH_SIZE_MAX**
      SPI flash size MAX

2.9.2 Bootloader Image Format

The bootloader image consists of the same structures as the application image, see Application Image Structures. The only difference is in the Bootloader Description structure.

To get information about the bootloader image, please run the following command:

```
esptool.py --chip esp32s2 image_info build/bootloader/bootloader.bin --version 2
```

File size: 26576 (bytes)
ESP32 image header
---------------------
Image version: 1
Entry point: 0x40080658
Segments: 4
Flash size: 2MB

(continues on next page)
Flash freq: 40m
Flash mode: D10

ESP32 extended image header

WP pin: 0xee
Flash pins drive settings: clk_drv: 0x0, q_drv: 0x0, d_drv: 0x0, cs0_drv: 0x0, hd_drv: 0x0, wp_drv: 0x0
Chip ID: 0
Minimal chip revision: v0.0, (legacy min_rev = 0)
Maximal chip revision: v3.99

Segments information

<table>
<thead>
<tr>
<th>Segment</th>
<th>Length</th>
<th>Load addr</th>
<th>File offs</th>
<th>Memory types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x01bb0</td>
<td>0x3fff0030</td>
<td>0x00000018</td>
<td>BYTE_ACCESSIBLE, DRAM, DIRAM_DRAM</td>
</tr>
<tr>
<td>2</td>
<td>0x03c90</td>
<td>0x40078000</td>
<td>0x00001bd0</td>
<td>CACHE_APP</td>
</tr>
<tr>
<td>3</td>
<td>0x00004</td>
<td>0x40080400</td>
<td>0x00005868</td>
<td>IRAM</td>
</tr>
<tr>
<td>4</td>
<td>0x00f2c</td>
<td>0x40080404</td>
<td>0x00005874</td>
<td>IRAM</td>
</tr>
</tbody>
</table>

ESP32 image footer

Checksum: 0x65 (valid)
Validation hash: 6f31a7f8512f26f6bce7c3b270f93bf6cf1ee4602c322998ca8ce27433527e92... (valid)

Bootloader information

Bootloader version: 1
ESP-IDF: v5.1-dev-4304-gcb51a3b-dirty
Compile time: Mar 30 2023 19:14:17

Bootloader Description

The DRAM0 segment of the bootloader binary starts with the `esp_bootloader_desc_t` structure which carries specific fields describing the bootloader. This structure is located at a fixed offset = `sizeof(esp_image_header_t) + sizeof(esp_image_segment_header_t)`.  

- `magic_byte` - the magic byte for the `esp_bootloader_desc` structure.
- `reserved` - reserved for the future IDF use.
- `version` - bootloader version, see `CONFIG_BOOTLOADER_PROJECT_VER`
- `idf_ver` - ESP-IDF version.
- `date_and_time` - compile date and time.
- `reserved2` - reserved for the future IDF use.

* - The maximum length is 32 characters, including null-termination character.

To get the `esp_bootloader_desc_t` structure from the running bootloader, use `esp_bootloader_get_description()`.

To get the `esp_bootloader_desc_t` structure from a running application, use `esp_ota_get_bootloader_description()`.

API Reference

Header File

- `components/esp_bootloader_format/include/esp_bootloader_desc.h`

This header file can be included with:
#include "esp_bootloader_desc.h"

- This header file is a part of the API provided by the `esp_bootloader_format` component. To declare that your component depends on `esp_bootloader_format`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_bootloader_format
```

or

```cmake
PRIV_REQUIRES esp_bootloader_format
```

## Functions

```c
const esp_bootloader_desc_t *esp_bootloader_get_description (void)
```

Return esp_bootloader_desc structure.

Intended for use by the bootloader.

**Returns**  Pointer to esp_bootloader_desc structure.

## Structures

```c
struct esp_bootloader_desc_t
```

Bootloader description structure.

### Public Members

- `uint8_t magic_byte`
  - Magic byte ESP_BOOTLOADER_DESC_MAGIC_BYTE

- `uint8_t reserved[3]`
  - reserved for IDF

- `uint32_t version`
  - Bootloader version

- `char idf_ver[32]`
  - Version IDF

- `char date_time[24]`
  - Compile date and time

- `uint8_t reserved2[16]`
  - reserved for IDF

### Macros

**ESP_BOOTLOADER_DESC_MAGIC_BYTE**

The magic byte for the esp_bootloader_desc structure that is in DRAM.

## 2.9.3 Application Level Tracing
Overview

ESP-IDF provides a useful feature for application behavior analysis called Application Level Tracing. The feature can be enabled in menuconfig and allows transfer of arbitrary data between the host and ESP32-S2 via JTAG interface with minimal overhead on program execution.

Developers can use this library to send application specific state of execution to the host, and receive commands or other types of information 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 SystemView.

API Reference

Header File

- components/app_trace/include/esp_app_trace.h
- This header file can be included with:

```c
#include "esp_app_trace.h"
```

- This header file is a part of the API provided by the app_trace component. To declare that your component depends on app_trace, add the following to your CMakeLists.txt:

```plaintext
REQUIRES app_trace
```

or

```plaintext
PRIV_REQUIRES app_trace
```

Functions

```c
esp_err_t esp_apptrace_init (void)
```

Initializes application tracing module.

**Note:** Should be called before any esp_apptrace_xxx call.

**Returns** ESP_OK on success, otherwise see esp_err_t

```c
void esp_apptrace_down_buffer_config (uint8_t *buf, uint32_t size)
```

Configures down buffer.

**Note:** Needs to be called before attempting to receive any data using esp_apptrace_down_buffer_get and esp_apptrace_read. 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.

```c
uint8_t *esp_apptrace_buffer_get (esp_apptrace_dest_t dest, uint32_t size, uint32_t tmo)
```

Allocates buffer for trace data. Once the data in the buffer is ready to be sent, esp_apptrace_buffer_put must be called to indicate it.

**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_APPTACE_TMO_INFINITE to wait indefinitely.

Returns non-NULL on success, otherwise NULL.

dest \( \times \) ptr \( \times \) tmo

\( \text{esp_err_t esp_apptrace_buffer_put (esp_apptrace_dest_t dest, uint8_t *ptr, uint32_t tmo)} \)

Indicates that the data in the buffer is ready to be sent. This function is a counterpart of and must be preceded by esp_apptrace_buffer_get.

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_APPTACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

dest \( \times \) data \( \times \) size \( \times \) tmo

\( \text{esp_err_t esp_apptrace_write (esp_apptrace_dest_t dest, const void *data, uint32_t size, uint32_t tmo)} \)

Writes data to trace buffer.

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_APPTACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

int \( \text{esp_apptrace_vprintf_to (esp_apptrace_dest_t dest, uint32_t tmo, const char *fmt, va_list ap)} \)

vprintf-like function to send log messages to host via specified HW interface.

Parameters
• dest – Indicates HW interface to send data.
• tmo – Timeout for operation (in us). Use ESP_APPTACE_TMO_INFINITE to wait indefinitely.
• fmt – Address of format string.
• ap – List of arguments.

Returns Number of bytes written.

int \( \text{esp_apptrace_vprintf (const char *fmt, va_list ap)} \)

vprintf-like function to send log messages to host.

Parameters
• fmt – Address of format string.
• ap – List of arguments.

Returns Number of bytes written.

dest \( \times \) tmo

\( \text{esp_err_t esp_apptrace_flush (esp_apptrace_dest_t dest, uint32_t tmo)} \)

Flushes remaining data in trace buffer to host.

Parameters
• dest – Indicates HW interface to flush data on.
• tmo – Timeout for operation (in us). Use ESP_APPTACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see esp_err_t

dest \( \times \) min_sz \( \times \) tmo

\( \text{esp_err_t esp_apptrace_flush_nolock (esp_apptrace_dest_t dest, uint32_t min_sz, uint32_t tmo)} \)

Flushes remaining data in trace buffer to host without locking internal data. This is a special version of esp_apptrace_flush which should be called from panic handler.

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.

Returns ESP_OK on success, otherwise see `esp_err_t`.

```c
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.

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.

Returns ESP_OK on success, otherwise see `esp_err_t`.

```c
uint8_t *esp_apptrace_down_buffer_get (esp_apptrace_dest_t dest, uint32_t *size, uint32_t tmo)
```

Retrieves incoming data buffer if any. Once data in the buffer is processed, `esp_apptrace_down_buffer_put` must be called to indicate it.

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.

Returns non-NULL on success, otherwise NULL.

```c
esp_err_t esp_apptrace_down_buffer_put (esp_apptrace_dest_t dest, uint8_t *ptr, uint32_t tmo)
```

Indicates that the data in the down buffer is processed. This function is a counterpart of and must be preceded by `esp_apptrace_down_buffer_get`.

Parameters
- `dest` - Indicates HW interface to receive data. Should be identical to the same parameter in call to `esp_apptrace_down_buffer_get`.
- `ptr` - Address of trace buffer to release. Should be the value returned by call to `esp_apptrace_down_buffer_get`.
- `tmo` - Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns ESP_OK on success, otherwise see `esp_err_t`.

```c
bool esp_apptrace_host_is_connected (esp_apptrace_dest_t dest)
```

Checks whether host is connected.

Parameters `dest` - Indicates HW interface to use.
Returns true if host is connected, otherwise false.

```c
void *esp_apptrace_fopen (esp_apptrace_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.

Parameters
- `dest` - Indicates HW interface to use.
- `path` - Path to file.
- `mode` - Mode string. See `fopen` for details.

Returns non zero file handle on success, otherwise 0

```c
int esp_apptrace_fclose (esp_apptrace_dest_t dest, void *stream)
```

Closes file on host. This function has the same semantic as `fclose` except for the first argument.

Parameters
- `dest` - Indicates HW interface to use.
Chapter 2. API Reference

- **stream** – File handle returned by esp_apptrace_fopen.

**Returns** Zero on success, otherwise non-zero. See fclose for details.

```c
size_t esp_apptrace_fwrite(esp_apptrace_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.

**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_apptrace_fopen.

**Returns** Number of written items. See fwrite for details.

```c
size_t esp_apptrace_fread(esp_apptrace_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.

**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_apptrace_fopen.

**Returns** Number of read items. See fread for details.

```c
int esp_apptrace_fseek(esp_apptrace_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.

**Parameters**
- **dest** – Indicates HW interface to use.
- **stream** – File handle returned by esp_apptrace_fopen.
- **offset** – Offset. See fseek for details.
- **whence** – Position in file. See fseek for details.

**Returns** Zero on success, otherwise non-zero. See fseek for details.

```c
int esp_apptrace_ftell(esp_apptrace_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.

**Parameters**
- **dest** – Indicates HW interface to use.
- **stream** – File handle returned by esp_apptrace_fopen.

**Returns** Current position in file. See ftell for details.

```c
int esp_apptrace_fstop(esp_apptrace_dest_t dest)
```

Indicates to the host that all file operations are complete. 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.).

**Parameters**
- **dest** – Indicates HW interface to use.

**Returns** ESP_OK on success, otherwise see esp_err_t

```c
int esp_apptrace_feof(esp_apptrace_dest_t dest, void *stream)
```

Test end-of-file indicator on a stream. This function has the same semantic as ‘feof’ except for the first argument.

**Parameters**
- **dest** – Indicates HW interface to use.
- **stream** – File handle returned by esp_apptrace_fopen.

**Returns** Non-Zero if end-of-file indicator is set for stream. See feof for details.
void esp_gcov_dump (void)

Triggers gcov info dump. This function waits for the host to connect to target before dumping data.

Enumerations

define esp_apptrace_dest_t

Application trace data destinations bits.

Values:

tenumerator ESP_APPTRACE_DEST_JTAG

JTAG destination.

tenumerator ESP_APPTRACE_DEST_TRAX

xxx_TRAX name is obsolete, use more common xxx_JTAG

tenumerator ESP_APPTRACE_DEST_UART

UART destination.

tenumerator ESP_APPTRACE_DEST_MAX

tenumerator ESP_APPTRACE_DEST_NUM

Header File

- components/app_trace/include/esp_sysview_trace.h

- This header file can be included with:

```c
#include "esp_sysview_trace.h"
```

- This header file is a part of the API provided by the app_trace component. To declare that your component depends on app_trace, add the following to your CMakeLists.txt:

```cmake
REQUIRES app_trace
```

or

```cmake
PRIV_REQUIRES app_trace
```

Functions

static inline esp_err_t esp_sysview_flush (uint32_t tmo)

Flushes remaining data in SystemView trace buffer to host.

Parameters

tmo - Timeout for operation (in us). Use ESP_APPTRACE_TMO_INFINITE to wait indefinitely.

Returns

ESP_OK.

int esp_sysview_vprintf (const char *format, va_list args)

vprintf-like function to sent log messages to the host.

Parameters

- format - Address of format string.
- args - List of arguments.

Returns

Number of bytes written.
Chapter 2. API Reference

**esp_err_t esp_sysview_heap_trace_start**(uint32_t tmo)

Starts SystemView heap tracing.

**Parameters**
- `tmo` - Timeout (in us) to wait for the host to be connected. Use -1 to wait forever.

**Returns**
- ESP_OK on success, ESP_ERR_TIMEOUT if operation has been timed out.

**esp_err_t esp_sysview_heap_trace_stop**(void)

Stops SystemView heap tracing.

**Returns**
- 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.

2.9.4 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 a 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 for that function
- the size of stack in bytes
- a pointer to the shared stack function

The user-defined function is deferred as a callback and can be called using the user-allocated space without taking space from current task stack.

The usage may look like the code below:

```c
void external_stack_function(void)
{
    printf("Executing this printf from external stack! \n");
}

//Let us suppose we want to call printf using a separated stack space
//allowing the app to reduce its stack size.
```

(continues on next page)
```c
void app_main()
{
    //Allocate a stack buffer, from heap or as a static form:
    StackType_t *shared_stack = malloc(8192 * sizeof(StackType_t));
    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**

- components/esp_system/include/esp_expression_with_stack.h
- This header file can be included with:

```c
#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 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)

**2.9.5 Chip Revision**
Overview

ESP32-S2 may have different revisions. These revisions mainly fix some issues, and sometimes also bring new features to the chip. Versioning Scheme describes the versioning of these chip revisions, and the APIs to read the versions at runtime.

There are some considerations of compatibility among application, ESP-IDF version, and chip revisions:

- Applications may depend on some fixes/features provided by a chip revision.
- When using updated version of hardware, the hardware may be incompatible with earlier versions of ESP-IDF.

Compatibility Checks of ESP-IDF describes how the application can specify its chip revision requirements, and the way ESP-IDF checks the compatibility. After that, there is troubleshooting information for this mechanism.

Versioning Scheme

A chip’s revision number is typically expressed as vX.Y, where:

- X means a Major wafer version. If it is changed, it means that the current software version is not compatible with this released chip and the software must be updated to use this chip.
- Y means a Minor wafer version. If it is changed that means the current software version is compatible with the released chip, and there is no need to update the software.

If a newly released chip does not contain breaking changes, the chip can run the same software as the previous chip. As such, the new chip’s revision number will only increment the minor version while keeping the major version the same (e.g., v1.1 to v1.2).

Conversely, if a newly released chip contains breaking changes, the chip cannot run the same software as the previous chip. As such, the new chip’s revision number will increment the major version and set the minor version to 0 (e.g., v1.1 to v2.0).

This versioning scheme was selected to indicate the derivation relationship of chip revisions, and clearly distinguish changes in chips between breaking changes and non-breaking changes.

ESP-IDF is designed to execute seamlessly on future chip minor revisions with the same logic as the chip’s nearest previous minor revision. Thus, users can directly port their compiled binaries to newer MINOR chip revisions without upgrading their ESP-IDF version and re-compile the whole project.

When a binary is executed on a chip revision of unexpected MAJOR revision, the software is also able to report issues according to the MAJOR revision. The major and minor versioning scheme also allows hardware changes to be branchable.

Note: The current chip revision scheme using major and minor versions was introduced from ESP-IDF v5.0 onwards. Thus bootloaders built using earlier versions of ESP-IDF will still use the legacy chip revision scheme of wafer versions.

EFuse Bits for Chip Revisions

Chips have several eFuse version fields:

- Major wafer version (WAFER_VERSION_MAJOR eFuse)
- Minor wafer version (WAFER_VERSION_MINOR eFuse)
- Ignore maximum revision (DISABLE_WAFER_VERSION_MAJOR eFuse). See Compatibility Checks of ESP-IDF on how this is used.

Note: The previous versioning logic was based on a single eFuse version field (WAFER_VERSION). This approach makes it impossible to mark chips as breaking or non-breaking changes, and the versioning logic becomes linear.
Chapter 2. API Reference

Chip Revision APIs These APIs helps to get chip revision from eFuses:

- `efuse_hal_chip_revision()`. It returns revision in the `major * 100 + minor` format.
- `efuse_hal_get_major_chip_version()`. It returns Major revision.
- `efuse_hal_get_minor_chip_version()`. It returns Minor revision.

The following Kconfig definitions (in `major * 100 + minor` format) that can help add the chip revision dependency to the code:

- `CONFIG_ESP32S2_REV_MIN_FULL`
- `CONFIG_ESP_REV_MIN_FULL`
- `CONFIG_ESP32S2_REV_MAX_FULL`
- `CONFIG_ESP_REV_MAX_FULL`

Compatibility Checks of ESP-IDF

When building an application that needs to support multiple revisions of a particular chip, the minimum and maximum chip revision numbers supported by the build are specified via Kconfig.

The minimum chip revision can be configured via the `CONFIG_ESP32S2_REV_MIN` option. Specifying the minimum chip revision will limit the software to only run on a chip revisions that are high enough to support some features or bugfixes.

The maximum chip revision cannot be configured and is automatically determined by the current ESP-IDF version being used. ESP-IDF will refuse to boot any chip revision exceeding the maximum chip revision. Given that it is impossible for a particular ESP-IDF version to foresee all future chip revisions, the maximum chip revision is usually set to maximum supported MAJOR version + 99. The “Ignore Maximum Revision” eFuse can be set to bypass the maximum revision limitation. However, the software is not guaranteed to work if the maximum revision is ignored.

Below is the information about troubleshooting when the chip revision fails the compatibility check. Then there are technical details of the checking and software behavior on earlier version of ESP-IDF.

Troubleshooting

1. If the 2nd stage bootloader is run on a chip revision smaller than minimum revision specified in the image (i.e., the application), a reboot occurs. The following message will be printed:

   **Image requires chip rev >= v3.0, but chip is v1.0**

To resolve this issue,

- Use a chip with the required minimum revision or higher.
- Lower the `CONFIG_ESP32S2_REV_MIN` value and rebuild the image so that it is compatible with the chip revision being used.

2. If application does not match minimum and maximum chip revisions, a reboot occurs. The following message will be printed:

   **Image requires chip rev <= v2.99, but chip is v3.0**

To resolve this issue, update ESP-IDF to a newer version that supports the chip’s revision (`CONFIG_ESP32S2_REV_MAX_FULL`). Alternatively, set the Ignore maximal revision bit in eFuse or use a chip revision that is compatible with the current version of ESP-IDF.

Representing Revision Requirements of a Binary Image The 2nd stage bootloader and the application binary images contain the `esp_image_header_t` header, which stores information specifying the chip revisions that the image is permitted to run on. This header has 3 fields related to revisions:

- `min_chip_rev` - Minimum chip MAJOR revision required by image (but for ESP32-C3 it is MINOR revision). Its value is determined by `CONFIG_ESP32S2_REV_MIN`.
• **min_chip_rev_full** - Minimum chip MINOR revision required by image in format: major * 100 + minor. Its value is determined by `CONFIG_ESP32S2_REV_MIN`.

• **max_chip_rev_full** - Maximum chip revision required by image in format: major * 100 + minor. Its value is determined by `CONFIG_ESP32S2_REV_MAX_FULL`. It can not be changed by user. Only Espressif can change it when a new version will be supported in ESP-IDF.

**Maximum And Minimum Revision Restrictions**  The order for checking the minimum and maximum revisions during application boot up is as follows:

1. The 1st stage bootloader (ROM bootloader) does not check minimum and maximum revision fields from `esp_image_header_t` before running the 2nd stage bootloader.

2. The initialization phase of the 2nd stage bootloader checks that the 2nd stage bootloader itself can be launched on the chip of this revision. It extracts the minimum revision from the header of the bootloader image and checks against the chip revision from eFuses. If the chip revision is less than the minimum revision, the bootloader refuses to boot up and aborts. The maximum revision is not checked at this phase.

3. Then the 2nd stage bootloader checks the revision requirements of the application. It extracts the minimum and maximum revisions from the header of the application image and checks against the chip revision from eFuses. If the chip revision is less than the minimum revision or higher than the maximum revision, the bootloader refuses to boot up and aborts. However, if the Ignore maximum revision bit is set, the maximum revision constraint can be ignored. The ignore bit is set by the customer themselves when there is confirmation that the software is able to work with this chip revision.

4. Furthermore, at the OTA update stage, the running application checks if the new software matches the chip revision. It extracts the minimum and maximum revisions from the header of the new application image and checks against the chip revision from eFuses. It checks for revision matching in the same way that the bootloader does, so that the chip revision is between the min and max revisions (logic of ignoring max revision also applies).

**Backward Compatibility with Bootloaders Built by Older ESP-IDF Versions**  The old bootloaders (ESP-IDF < v5.0) do not know about Major and Minor wafer version eFuses. They use one single eFuse for this - wafer version.

ESP32-S2 chip support was added in ESP-IDF v4.2. ESP32-S2 chips have `rev_min` in `esp_image_header_t` header set to 0 because Minimum Supported ESP32-S2 Revision Kconfig option was not introduced, which means that the old bootloader does not check the chip revision. Any app can be loaded by such bootloader in range v0.0 to v3.15.

Please check the chip version using `esptool chip_id` command.

**References**

- Compatibility Advisory for Chip Revision Numbering Scheme
- Compatibility Between ESP-IDF Releases and Revisions of Espressif SoCs
- SoC Errata
- ESP-IDF Versions

**API Reference**

**Header File**

- `components/hal/include/hal/efuse_hal.h`

This header file can be included with:

```c
#include "hal/efuse_hal.h"
```

**Functions**

```c
void efuse_hal_get_mac (uint8_t *mac)
get factory mac address
```
```c
uint32_t efuse_hal_chip_revision (void)
    Returns chip version.

    Returns Chip version in format: Major * 100 + Minor

uint32_t efuse_hal_blk_version (void)
    Return block version.

    Returns Block version in format: Major * 100 + Minor

bool efuse_hal_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.

    Returns true if flash encryption is enabled.

bool efuse_hal_get_disable_wafer_version_major (void)
    Returns the status of whether the bootloader (and OTA) will check the maximum chip version or not.

    Returns true - Skip the maximum chip version check.

uint32_t efuse_hal_get_major_chip_version (void)
    Returns major chip version.

uint32_t efuse_hal_get_minor_chip_version (void)
    Returns minor chip version.
```

## 2.9.6 Console

ESP-IDF provides console component, which includes building blocks needed to develop an interactive console over serial port. This component includes the following features:

- 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 used for parsing GNU style command line arguments.
- Functions for registration and dispatching of commands.
- Functions to establish a basic REPL (Read-Evaluate-Print-Loop) environment.

**Note:** These features 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.

**Note:** When using a console application on a chip that supports a hardware USB serial interface, we suggest to disable the secondary serial console output. The secondary output will be output-only and consequently does not make sense in an interactive application.

### Line Editing

Line editing feature lets users compose commands by typing them, erasing symbols using the backspace key, navigating within the command using the left/right keys, navigating to previously typed commands using the up/down keys, and performing autocompletion using the tab key.
Chapter 2. API Reference

Note: This feature relies on ANSI escape sequence support in the terminal application. As such, serial monitors which display raw UART data cannot be used together with the line editing library. If you see \[6n or similar escape sequence when running system/console example instead of a command prompt (e.g., esp>), it means that the serial monitor does not support escape sequences. Programs which are known to work are GNU screen, minicom, and esp-idf-monitor (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 mode needs less data to be sent to refresh screen on each keypress, 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.
- linenoiseAllowEmpty()  
  Set whether linenoise library returns a zero-length string (if true) or NULL (if false) for empty lines. By default, zero-length strings are returned.
- linenoiseSetMaxLineLen()  
  Set maximum length of the line for linenoise library. Default length is 4096 bytes. The default value can be updated to optimize RAM memory usage.

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 the completed line once the 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 the user presses the tab key, linenoise library invokes the 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 the 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 the 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 the up/down arrows keys.
- `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.

Splitting of Command Line into Arguments

The `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\"]`

Argument Parsing

For argument parsing, the `console` component includes `argtable3` library. Please see the `tutorial` for an introduction to `argtable3`. Github repository also includes examples.

Command Registration and Dispatching

The `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.

**Initialize Console REPL Environment**

To establish a basic REPL environment, `console` component provides several useful APIs, combining those functions described above.

In a typical application, you only need to call `esp_console_new_repl_uart()` to initialize the REPL environment based on UART device, including driver install, basic console configuration, spawning a thread to do REPL task and register several useful commands (e.g., `help`).

After that, you can register your own commands with `esp_console_cmd_register()`. The REPL environment keeps in init state until you call `esp_console_start_repl()`.

**Application Example**

Example application illustrating usage of the `console` component is available in `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.

Besides that, ESP-IDF contains several useful examples which are based on the `console` component and can be treated as “tools” when developing applications. For example, `peripherals/i2c/i2c_tools`, `wifi/iperf`.

**API Reference**

**Header File**

- **components/console/esp_console.h**
  This header file can be included with:

  ```
  #include "esp_console.h"
  ```

  This header file is a part of the API provided by the `console` component. To declare that your component depends on `console`, add the following to your CMakeLists.txt:

  ```
  REQUIRE console
  ```
  or

  ```
  PRIV_REQUIRES console
  ```

**Functions**
**esp_err_t esp_console_init** (const *esp_console_config_t* config)

initialize console module

**Note:** Call this once before using other console module features

**Parameters**

- config – console configuration

**Returns**

- ESP_OK on success
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_STATE if already initialized
- ESP_ERR_INVALID_ARG if the configuration is invalid

**esp_err_t esp_console_deinit** (void)

de-initialize console module

**Note:** Call this once when done using console module functions

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_STATE if not initialized yet

**esp_err_t esp_console_cmd_register** (const *esp_console_cmd_t* cmd)

Register console command.

**Parameters**

- cmd – pointer to the command description; can point to a temporary value

**Returns**

- ESP_OK on success
- ESP_ERR_NO_MEM if out of memory
- ESP_ERR_INVALID_ARG if command description includes invalid arguments

**esp_err_t esp_console_run** (const char* cmdline, int* cmd_ret)

Run command line.

**Parameters**

- cmdline – command line (command name followed by a number of arguments)
- cmd_ret – [out] return code from the command (set if command was run)

**Returns**

- ESP_OK, if command was run
- ESP_ERR_INVALID_ARG, if the command line is empty, or only contained whitespace
- ESP_ERR_NOT_FOUND, if command with given name wasn’t registered
- ESP_ERR_INVALID_STATE, if esp_console_init wasn’t called

**size_t esp_console_split_argv** (char* line, char** argv, size_t argv_size)

Split command line into arguments in place.

* - This function finds whitespace-separated arguments in the given input line.
* - 'abc def 1 20 .3' -> ['abc', 'def', '1', '20', '.3']
* - Argument which include spaces may be surrounded with quotes. In this case spaces are preserved and quotes are stripped.
* - 'abc "123 456" def' -> ['abc', '123 456', 'def']
* - Escape sequences may be used to produce backslash, double quote, and space; (continues on next page)
Note: Pointers to at most argv_size - 1 arguments are returned in argv array. The pointer after the last one (i.e. argv[argc]) is set to NULL.

Parameters
- line - pointer to buffer to parse; it is modified in place
- argv - array where the pointers to arguments are written
- argv_size - number of elements in argv_array (max. number of arguments)

Returns number of arguments found (argc)

void esp_console_get_completion (const char *buf, linenoiseCompletions *lc)
Callback which provides command completion for linenoise library.

When using linenoise for line editing, command completion support can be enabled like this:
linenoiseSetCompletionCallback(&esp_console_get_completion);

Parameters
- buf - the string typed by the user
- lc - linenoiseCompletions to be filled in

const char *esp_console_get_hint (const char *buf, int *color, int *bold)
Callback which provides command hints for linenoise library.

When using linenoise for line editing, hints support can be enabled as follows:
linenoiseSetHintsCallback((linenoiseHintsCallback *)&esp_console_get_hint);

The extra cast is needed because linenoiseHintsCallback is defined as returning a char* instead of const char*.

Parameters
- buf - line typed by the user
- color - [out] ANSI color code to be used when displaying the hint
- bold - [out] set to 1 if hint has to be displayed in bold

Returns string containing the hint text. This string is persistent and should not be freed (i.e. linenoiseSetFreeHintsCallback should not be used).

esp_err_t esp_console_register_help_command (void)
Register a ‘help’ command.

Default ‘help’ command prints the list of registered commands along with hints and help strings if no additional argument is given. If an additional argument is given, the help command will look for a command with the same name and only print the hints and help strings of that command.

Returns
- ESP_OK on success
- ESP_ERR_INVALID_STATE, if esp_console_init wasn’t called

esp_err_t esp_console_new_repl_uart (const esp_console_dev_uart_config_t *dev_config, const esp_console_repl_config_t *repl_config, esp_console_repl_t **ret_repl)
Establish a console REPL environment over UART driver.

Attention This function is meant to be used in the examples to make the code more compact. Applications which use console functionality should be based on the underlying linenoise and esp_console functions.
Note: This is an all-in-one function to establish the environment needed for REPL, includes:

- Install the UART driver on the console UART (8n1, 115200, REF_TICK clock source)
- Configures the stdin/stdout to go through the UART driver
- Initializes linenoise
- Spawn new thread to run REPL in the background

Parameters

- `dev_config` [in] UART device configuration
- `repl_config` [in] REPL configuration
- `ret_repl` [out] return REPL handle after initialization succeed, return NULL otherwise

Returns

- ESP_OK on success
- ESP_FAIL Parameter error

```c
esp_err_t esp_console_new_repl_usb_cdc(const esp_console_dev_usb_cdc_config_t *dev_config, const esp_console_repl_config_t *repl_config, esp_console_repl_t **ret_repl)
```

Establish a console REPL environment over USBCDC.

Attention This function is meant to be used in the examples to make the code more compact. Applications which use console functionality should be based on the underlying linenoise and esp_console functions.

Note: This is a all-in-one function to establish the environment needed for REPL, includes:

- Initializes linenoise
- Spawn new thread to run REPL in the background

Parameters

- `dev_config` [in] USBCDC configuration
- `repl_config` [in] REPL configuration
- `ret_repl` [out] return REPL handle after initialization succeed, return NULL otherwise

Returns

- ESP_OK on success
- ESP_FAIL Parameter error

```c
esp_err_t esp_console_start_repl(esp_console_repl_t *repl)
```

Start REPL environment.

Note: Once the REPL gets started, it won’t be stopped until the user calls `repl->del(repl)` to destroy the REPL environment.

Parameters `repl` [in] REPL handle returned from `esp_console_new_repl_xxx`

Returns

- ESP_OK on success
- ESP_ERR_INVALID_STATE, if repl has started already

**Structures**
struct esp_console_config_t
Parameters for console initialization.

Public Members

size_t max_cmdline_length
length of command line buffer, in bytes

size_t max_cmdline_args
maximum number of command line arguments to parse

uint32_t heap_alloc_caps
where to (e.g. MALLOC_CAP_SPIRAM) allocate heap objects such as cmds used by esp_console

int hint_color
ASCII color code of hint text.

int hint_bold
Set to 1 to print hint text in bold.

struct esp_console_repl_config_t
Parameters for console REPL (Read Eval Print Loop)

Public Members

uint32_t max_history_len
maximum length for the history

const char *history_save_path
file path used to save history commands, set to NULL won’t save to file system

uint32_t task_stack_size
repl task stack size

uint32_t task_priority
repl task priority

const char *prompt
prompt (NULL represents default: “esp> “)

size_t max_cmdline_length
maximum length of a command line. If 0, default value will be used

struct esp_console_dev_uart_config_t
Parameters for console device: UART.
**Public Members**

`int channel`

UART channel number (count from zero)

`int baud_rate`

Communication baud rate.

`int tx_gpio_num`

GPIO number for TX path, -1 means using default one.

`int rx_gpio_num`

GPIO number for RX path, -1 means using default one.

`struct esp_console_dev_usb_cdc_config_t`

Parameters for console device: USB CDC.

---

**Note:** It’s an empty structure for now, reserved for future

---

`struct esp_console_cmd_t`

Console command description.

**Public Members**

`const char *command`

Command name. Must not be NULL, must not contain spaces. The pointer must be valid until the call to `esp_console_deinit`.

`const char *help`

Help text for the command, shown by `help` command. If set, the pointer must be valid until the call to `esp_console_deinit`. If not set, the command will not be listed in `help` output.

`const char *hint`

Hint text, usually lists possible arguments. If set to NULL, and ‘argtable’ field is non-NULL, hint will be generated automatically.

`esp_console_cmd_func_t func`

Pointer to a function which implements the command.

`void *argtable`

Array or structure of pointers to arg_xxx structures, may be NULL. Used to generate hint text if ‘hint’ is set to NULL. Array/structure which this field points to must end with an arg_end. Only used for the duration of `esp_console_cmd_register` call.

`struct esp_console_repl_s`

Console REPL base structure.
Public Members

```c
esp_err_t (*del)(esp_console_repl_t *repl)
```
Delete console REPL environment.

**Param repl** [in] REPL handle returned from esp_console_new_repl_xxx

**Return**
- ESP_OK on success
- ESP_FAIL on errors

**Macros**

```c
ESP_CONSOLE_CONFIG_DEFAULT ()
```
Default console configuration value.

```c
ESP_CONSOLE_REPL_CONFIG_DEFAULT ()
```
Default console repl configuration value.

```c
ESP_CONSOLE_DEV_UART_CONFIG_DEFAULT ()
```

```c
ESP_CONSOLE_DEV_CDC_CONFIG_DEFAULT ()
```

**Type Definitions**

```c
typedef struct linenoiseCompletions linenoiseCompletions
```

```c
typedef int (*esp_console_cmd_func_t)(int argc, char **argv)
```
Console command main function.

**Param argc** number of arguments

**Param argv** array with argc entries, each pointing to a zero-terminated string argument

**Return** console command return code, 0 indicates “success”

```c
typedef struct esp_console_repl_s esp_console_repl_t
```
Type defined for console REPL.

### 2.9.7 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-S2 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-S2 Technical Reference Manual > eFuse Controller (eFuse) [PDF]. Some eFuse bits are available for user applications.

ESP32-S2 has 11 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 entirely for system purposes;
• EFUSE_BLK2 is used entirely for system purposes;
• EFUSE_BLK3 (also named EFUSE_BLK_USER_DATA) can be used for user purposes;
• EFUSE_BLK4 (also named EFUSE_BLK_KEY0) can be used as key (for secure_boot or flash_encryption) or for user purposes;
• EFUSE_BLK5 (also named EFUSE_BLK_KEY1) can be used as key (for secure_boot or flash_encryption) or for user purposes;
• EFUSE_BLK6 (also named EFUSE_BLK_KEY2) can be used as key (for secure_boot or flash_encryption) or for user purposes;
• EFUSE_BLK7 (also named EFUSE_BLK_KEY3) can be used as key (for secure_boot or flash_encryption) or for user purposes;
• EFUSE_BLK8 (also named EFUSE_BLK_KEY4) can be used as key (for secure_boot or flash_encryption) or for user purposes;
• EFUSE_BLK9 (also named EFUSE_BLK_KEY5) can be used as key (for secure_boot or flash_encryption) or for user purposes;
• EFUSE_BLK10 (also named EFUSE_BLK_SYS_DATA_PART2) is reserved for system purposes.

Each block is divided into 8 32-bit 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 checks with the **common** CSV file.

CSV files:

- **common** *(esp_efuse_table.csv)* - contains eFuse fields which are used inside the ESP-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:

```csv
# field_name, efuse_block(EFUSE_BLK0..EFUSE_BLK10), 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_` is added to the name, and this field name is available in the code. This name is 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). The field-name supports structured format using . to show that the field belongs to another field (see `WR_DIS` and `RD_DIS` in the common table).

- **efuse_block**
  
  Block number. It determines where the eFuse bits are placed for this field. Available EFUSE_BLK0..EFUSE_BLK10.
Chapter 2. API Reference

bit_start

Start bit number (0..255). The bit_start field can be omitted. In this case, it is 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 cannot be omitted. This field also may be MAX_BLK_LEN in this case, the field length has the maximum block length.

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, indicating that it belongs to one field. For example two fields MAC_FACTORY and MAC_FACTORY_CRC:

```
# Factory MAC address #
MAC_FACTORY, 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, 80, 8, CRC8 for factory MAC address
```

This field is available in code as ESP_EFUSE_MAC_FACTORY and ESP_EFUSE_MAC_FACTORY_CRC.

Structured eFuse Fields

```
WR_DIS, EFUSE_BLK0, 0, 32, Write protection
WR_DIS.RD_DIS, EFUSE_BLK0, 0, 1, Write protection for...
 → RD_DIS
WR_DIS.FIELD_1, EFUSE_BLK0, 1, 1, Write protection for...
 → FIELD_1
WR_DIS.FIELD_2, EFUSE_BLK0, 2, 4, Write protection for...
 → FIELD_2 (includes B1 and B2)
WR_DIS.FIELD_2.B1, EFUSE_BLK0, 2, 2, Write protection for...
 → FIELD_2.B1
WR_DIS.FIELD_2.B2, EFUSE_BLK0, 4, 2, Write protection for...
 → FIELD_2.B2
WR_DIS.FIELD_3, EFUSE_BLK0, 5, 1, Write protection for...
 → FIELD_3
WR_DIS.FIELD_3.ALIAS, EFUSE_BLK0, 5, 1, Write protection for...
 → FIELD_3 (just a alias for WR_DIS.FIELD_3)
WR_DIS.FIELD_4, EFUSE_BLK0, 7, 1, Write protection for...
 → FIELD_4
```

The structured eFuse field looks like WR_DIS.RD_DIS where the dot points that this field belongs to the parent field - WR_DIS and cannot be out of the parent’s range.

It is possible to use some levels of structured fields as WR_DIS.FIELD_2.B1 and B2. These fields should not be crossed each other and should be in the range of two fields: WR_DIS and WR_DIS.FIELD_2.

It is possible to create aliases for fields with the same range, see WR_DIS.FIELD_3 and WR_DIS.FIELD_3.ALIAS.

The ESP-IDF names for structured eFuse fields should be unique. The efuse_table_gen tool generates the final names where the dot is replaced by _. The names for using in ESP-IDF are ESP_EFUSE_WR_DIS, ESP_EFUSE_WR_DIS_RD_DIS, ESP_EFUSE_WR_DIS_FIELD_2_B1, etc.
The `efuse_table_gen` tool checks that the fields do not overlap each other and must be within the range of a field if there is a violation, then throws the following error:

```
Field at USER_DATA, EFUSE_BLK3, 0, 256 intersected with SERIAL_NUMBER, EFUSE_BLK3, 0, 32
```

Solution: Describe SERIAL_NUMBER to be included in USER_DATA.(USER_DATA.SERIAL_NUMBER).

```
Field at FIELD, EFUSE_BLK3, 0, 50 out of range FIELD.MAJOR_NUMBER, EFUSE_BLK3, 0, 60
```

Solution: Change bit_start for FIELD.MAJOR_NUMBER from 60 to 0, so MAJOR_NUMBER is in the FIELD range.

**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 --idf_target esp32s2 esp32s2/esp_efuse_table.csv
```

After generation in the folder `$IDF_PATH/components/efuse/esp32s2` 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 --idf_target esp32s2 esp32s2/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"
```

**Supported Coding Scheme**

Coding schemes are used to protect against data corruption. ESP32-S2 supports two coding schemes:

- **None.** EFUSE_BLK0 is stored with four backups, meaning each bit is stored four times. This backup scheme is automatically applied by the hardware and is not visible to software. EFUSE_BLK0 can be written many times.
- **RS.** EFUSE_BLK1 - EFUSE_BLK10 use Reed-Solomon coding scheme that supports up to 5 bytes of automatic error correction. Software encodes the 32-byte EFUSE_BLKx using RS (44, 32) to generate a 12-byte check code, and then burn the EFUSE_BLKx and the check code into eFuse at the same time. The eFuse Controller automatically decodes the RS encoding and applies error correction when reading back the eFuse block. Because the RS check codes are generated across the entire 256-bit eFuse block, each block can only be written to one time.
To write some fields into one block, or different blocks in one time, you need to use the batch writing mode. Firstly set this mode through `esp_efuse_batch_write_begin()` function then write some fields as usual using the `esp_efuse_write_...` functions. At the end to burn them, call the `esp_efuse_batch_write_commit()` function. It burns prepared data to the eFuse blocks and disables the batch recording mode.

**Note:** If there is already pre-written data in the eFuse block using the Reed-Solomon encoding scheme, then it is not possible to write anything extra (even if the required bits are empty) without breaking the previous encoding data. This encoding data will be overwritten with new encoding data and completely destroyed (however, the payload eFuses are not damaged). It can be related to: CUSTOM_MAC, SPI_PAD_CONFIG_HD, SPI_PAD_CONFIG_CS, etc. Please contact Espressif to order the required pre-burnt eFuses.

FOR TESTING ONLY (NOT RECOMMENDED): You can ignore or suppress errors that violate encoding scheme data in order to burn the necessary bits in the eFuse block.

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()` - sets 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.
- `esp_efuse_get_key_dis_read()` - Returns a read protection for the key block.
- `esp_efuse_set_key_dis_read()` - Sets a read protection for the key block.
- `esp_efuse_get_key_dis_write()` - Returns a write protection for the key block.
- `esp_efuse_set_key_dis_write()` - Sets a write protection for the key block.
- `esp_efuse_get_key_purpose()` - Returns the current purpose set for an eFuse key block.
- `esp_efuse_write_key()` - Programs a block of key data to an eFuse block
- `esp_efuse_write_keys()` - Programs keys to unused eFuse blocks
- `esp_efuse_find_purpose()` - Finds a key block with the particular purpose set.
- `esp_efuse_get_keypurpose_dis_write()` - Returns a write protection of the key purpose field for an eFuse key block (for esp32 always true).
- `esp_efuse_key_block_unused()` - Returns true if the key block is unused, false otherwise.

For frequently used fields, special functions are made, like this `esp_efuse_get_pkg_ver()`.

eFuse API for Keys

EFUSE_BLK_KEY0 - EFUSE_BLK_KEY5 are intended to keep up to 6 keys with a length of 256-bits. Each key has an ESP_EFUSE_KEY_PURPOSE_x field which defines the purpose of these keys. The purpose field is described in `esp_efuse_purpose_t`.

The purposes like `ESP_EFUSE_KEY_PURPOSE_XTS_AES...` are used for flash encryption.

The purposes like `ESP_EFUSE_KEY_PURPOSE_SECURE_BOOT_DIGEST...` are used for secure boot.

There are some eFuse APIs useful to work with states of keys.
• `esp_efuse_get_purpose_field()` - Returns a pointer to a key purpose for an eFuse key block.
• `esp_efuse_get_key()` - Returns a pointer to a key block.
• `esp_efuse_set_key_purpose()` - Sets a key purpose for an eFuse key block.
• `esp_efuse_set_key_purpose_dis_write()` - Sets a write protection of the key purpose field for an eFuse key block.
• `esp_efuse_find_unused_key_block()` - Search for an unused key block and return the first one found.
• `esp_efuse_count_unused_key_blocks()` - Returns the number of unused eFuse key blocks in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
• `esp_efuse_get_digest_revoke()` - Returns the status of the Secure Boot public key digest revocation bit.
• `esp_efuse_set_digest_revoke()` - Sets the Secure Boot public key digest revocation bit.
• `esp_efuse_get_write_protect_of_digest_revoke()` - Returns a write protection of the Secure Boot public key digest revocation bit.
• `esp_efuse_set_write_protect_of_digest_revoke()` - Sets a write protection of the Secure Boot public key digest revocation bit.

How to 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 esp32s2/esp_efuse_table.csv --info
```

Parsing efuse CSV input file $IDF_PATH/components/efuse/esp32s2/esp_efuse_table.csv...
Verifying efuse table...
Max number of bits in BLK 256
Sorted efuse table:
```
# field_name    efuse_block    bit_start  bit_count
 1  WR_DIS       EFUSE_BLK0    0          32
 2  WR_DIS.RD_DIS EFUSE_BLK0    0          1
 3  WR_DIS.DIS_ICACHE EFUSE_BLK0 2          1
 4  WR_DIS.DIS_DCACHE EFUSE_BLK0 2          1
 5  WR_DIS.DIS_DOWNLOAD_ICACHE EFUSE_BLK0 2          1
 6  WR_DIS.DIS_DOWNLOAD_DCACHE EFUSE_BLK0 2          1
 7  WR_DIS.DIS_FORCE_DOWNLOAD EFUSE_BLK0 2          1
 8  WR_DIS.DIS_USB EFUSE_BLK0    2          1
 9  WR_DIS.TWAI EFUSE_BLK0    2          1
10  WR_DIS.DIS_BOOT_REMAP EFUSE_BLK0 2          1
11  WR_DIS.SOFT_DIS_JTAG EFUSE_BLK0 2          1
12  WR_DIS.HARD_DIS_JTAG EFUSE_BLK0 2          1
13  WR.DIS.DIS_DOWNLOAD_MANUAL_ENCRYPT EFUSE_BLK0 2  ...
    ...
```

(continues on next page)
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Block</th>
<th>Offset</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>WR_DIS.SECURE_BOOT_AGGRESSIVE_REVOKE</td>
<td>EFUSE_BLK0</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>WR_DIS.FLASH_TPUW</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>WR_DIS.DIS_DOWNLOAD_MODE</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>WR_DIS.DIS_LEGACY_SPI_BOOT</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>WR_DIS.DIS_UART_PRINT_CHANNEL</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>WR_DIS.DIS_USB_DOWNLOAD_MODE</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>WR_DIS.ENABLE_SECURITY_DOWNLOAD</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>WR_DIS.UART_PRINT_CONTROL</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>WR_DIS.PIN_POWER_SELECTION</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>WR_DIS.FLASH_TYPE</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>WR_DIS.FORCE_SEND_RESUME</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>WR_DIS.SECURE_VERSION</td>
<td>EFUSE_BLK0</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>WR_DIS.BLK1</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>WR_DIS.MAC</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>WR_DIS.SPI_PAD_CONFIG_CLK</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>WR_DIS.SPI_PAD_CONFIG_Q</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>WR_DIS.SPI_PAD_CONFIG_D</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>WR_DIS.SPI_PAD_CONFIG_CS</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>WR_DIS.SPI_PAD_CONFIG_Q2</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>WR_DIS.SPI_PAD_CONFIG_W</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>WR_DIS.SPI_PAD_CONFIG/Q</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>WR_DIS.SPI_PAD_CONFIG_Q2</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>WR_DIS.SPI_PAD_CONFIG_Q3</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>WR_DIS.SPI_PAD_CONFIG_D3</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>WR_DIS.SPI_PAD_CONFIG_D7</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>54</td>
<td>WR_DIS.WAFER_VERSION_MAJOR</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>WR_DIS.WAFER_VERSION_MINOR_HI</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>56</td>
<td>WR_DIS.FLASH_VERSION</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>57</td>
<td>WR_DIS.BLK_VERSION_MAJOR</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>58</td>
<td>WR_DIS.PSRAM_VERSION</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>59</td>
<td>WR_DIS.PKG_VERSION</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>WR_DIS.WAFER_VERSION_MINOR_LO</td>
<td>EFUSE_BLK0</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>61</td>
<td>WR_DIS.SYS_DATA_PART1</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>62</td>
<td>WR_DIS.OPTIONAL_UNIQUE_ID</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>63</td>
<td>WR_DIS.ADC_CALIB</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>WR_DIS.BLK_VERSION_MINOR</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>65</td>
<td>WR_DIS.TEMP_CALIB</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>66</td>
<td>WR_DIS.RTCCALIB_V1IDX_A10H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>67</td>
<td>WR_DIS.RTCCALIB_V1IDX_A11H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>WR_DIS.RTCCALIB_V1IDX_A12H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>69</td>
<td>WR_DIS.RTCCALIB_V1IDX_A13H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>70</td>
<td>WR_DIS.RTCCALIB_V1IDX_A20H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>71</td>
<td>WR_DIS.RTCCALIB_V1IDX_A21H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>72</td>
<td>WR_DIS.RTCCALIB_V1IDX_A22H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>73</td>
<td>WR_DIS.RTCCALIB_V1IDX_A23H</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>74</td>
<td>WR_DIS.RTCCALIB_V1IDX_A10L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>75</td>
<td>WR_DIS.RTCCALIB_V1IDX_A11L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>76</td>
<td>WR_DIS.RTCCALIB_V1IDX_A12L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>77</td>
<td>WR_DIS.RTCCALIB_V1IDX_A13L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>78</td>
<td>WR_DIS.RTCCALIB_V1IDX_A20L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>79</td>
<td>WR_DIS.RTCCALIB_V1IDX_A21L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>WR_DIS.RTCCALIB_V1IDX_A22L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>81</td>
<td>WR_DIS.RTCCALIB_V1IDX_A23L</td>
<td>EFUSE_BLK0</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>82</td>
<td>WR_DIS.BLOCK_USR_DATA</td>
<td>EFUSE_BLK0</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>83</td>
<td>WR_DIS.CUSTOM_MAC</td>
<td>EFUSE_BLK0</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>84</td>
<td>WR_DIS.UART_PRINT</td>
<td>EFUSE_BLK0</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>85</td>
<td>WR_DIS.BLOCK_KEY0</td>
<td>EFUSE_BLK0</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>86</td>
<td>WR_DIS.BLOCK_KEY1</td>
<td>EFUSE_BLK0</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>87</td>
<td>WR_DIS.BLOCK_KEY2</td>
<td>EFUSE_BLK0</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>Block</td>
<td>Offset</td>
<td>Size</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------</td>
<td>------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>88</td>
<td>WR_DIS.BLOCK_KEY4</td>
<td>EFUSE_BLK0</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>89</td>
<td>WR_DIS.BLOCK_KEY5</td>
<td>EFUSE_BLK0</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>WR_DIS.BLOCK_SYS_DATA2</td>
<td>EFUSE_BLK0</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>91</td>
<td>WR_DIS.USB_EXCHG_PINS</td>
<td>EFUSE_BLK0</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>92</td>
<td>WR_DIS.USB_EXT_PHY_ENABLE</td>
<td>EFUSE_BLK0</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>WR_DIS.USB_FORCE_NOPERSIST</td>
<td>EFUSE_BLK0</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>94</td>
<td>WR_DIS.BLOCK0_VERSION</td>
<td>EFUSE_BLK0</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>95</td>
<td>RD_DIS</td>
<td>EFUSE_BLK0</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>96</td>
<td>RD_DIS.BLOCK_KEY0</td>
<td>EFUSE_BLK0</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>97</td>
<td>RD_DIS.BLOCK_KEY1</td>
<td>EFUSE_BLK0</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>98</td>
<td>RD_DIS.BLOCK_KEY2</td>
<td>EFUSE_BLK0</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>99</td>
<td>RD_DIS.BLOCK_KEY3</td>
<td>EFUSE_BLK0</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>RD_DIS.BLOCK_KEY4</td>
<td>EFUSE_BLK0</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>RD_DIS.BLOCK_KEY5</td>
<td>EFUSE_BLK0</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>RD_DIS.BLOCK_SYS_DATA2</td>
<td>EFUSE_BLK0</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>103</td>
<td>DIS_ICACHE</td>
<td>EFUSE_BLK0</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>104</td>
<td>DIS_DCACHE</td>
<td>EFUSE_BLK0</td>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>105</td>
<td>DIS_DOWNLOAD_ICACHE</td>
<td>EFUSE_BLK0</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>106</td>
<td>DIS_DOWNLOAD_DCACHE</td>
<td>EFUSE_BLK0</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>107</td>
<td>DIS_FORCE_DOWNLOAD</td>
<td>EFUSE_BLK0</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>108</td>
<td>DIS_USB</td>
<td>EFUSE_BLK0</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>109</td>
<td>DIS_TWAI</td>
<td>EFUSE_BLK0</td>
<td>46</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>DIS_BOOT_REMAP</td>
<td>EFUSE_BLK0</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>SOFT_DIS_JTAG</td>
<td>EFUSE_BLK0</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>112</td>
<td>HARD_DIS_JTAG</td>
<td>EFUSE_BLK0</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>113</td>
<td>DIS_DOWNLOAD_MANUAL_ENCRYPT</td>
<td>EFUSE_BLK0</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>114</td>
<td>USB_EXCHG_PINS</td>
<td>EFUSE_BLK0</td>
<td>56</td>
<td>1</td>
</tr>
<tr>
<td>115</td>
<td>USB_EXT_PHY_ENABLE</td>
<td>EFUSE_BLK0</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>116</td>
<td>USB_FORCE_NOPERSIST</td>
<td>EFUSE_BLK0</td>
<td>58</td>
<td>1</td>
</tr>
<tr>
<td>117</td>
<td>BLOCK0_VERSION</td>
<td>EFUSE_BLK0</td>
<td>59</td>
<td>2</td>
</tr>
<tr>
<td>118</td>
<td>VDD_SPI_XPD</td>
<td>EFUSE_BLK0</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>119</td>
<td>VDD_SPI_TIEH</td>
<td>EFUSE_BLK0</td>
<td>69</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>VDD_SPI_FORCE</td>
<td>EFUSE_BLK0</td>
<td>70</td>
<td>1</td>
</tr>
<tr>
<td>121</td>
<td>WDT_DELAY_SEL</td>
<td>EFUSE_BLK0</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>122</td>
<td>SPI_BOOT_CRYPT_CNT</td>
<td>EFUSE_BLK0</td>
<td>82</td>
<td>3</td>
</tr>
<tr>
<td>123</td>
<td>SECURE_BOOT_KEY_REVOKE0</td>
<td>EFUSE_BLK0</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>124</td>
<td>SECURE_BOOT_KEY_REVOKE1</td>
<td>EFUSE_BLK0</td>
<td>86</td>
<td>1</td>
</tr>
<tr>
<td>125</td>
<td>SECURE_BOOT_KEY_REVOKE2</td>
<td>EFUSE_BLK0</td>
<td>87</td>
<td>1</td>
</tr>
<tr>
<td>126</td>
<td>KEY_PURPOSE_0</td>
<td>EFUSE_BLK0</td>
<td>88</td>
<td>4</td>
</tr>
<tr>
<td>127</td>
<td>KEY_PURPOSE_1</td>
<td>EFUSE_BLK0</td>
<td>92</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>KEY_PURPOSE_2</td>
<td>EFUSE_BLK0</td>
<td>96</td>
<td>4</td>
</tr>
<tr>
<td>129</td>
<td>KEY_PURPOSE_3</td>
<td>EFUSE_BLK0</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>130</td>
<td>KEY_PURPOSE_4</td>
<td>EFUSE_BLK0</td>
<td>104</td>
<td>4</td>
</tr>
<tr>
<td>131</td>
<td>KEY_PURPOSE_5</td>
<td>EFUSE_BLK0</td>
<td>108</td>
<td>4</td>
</tr>
<tr>
<td>132</td>
<td>SECURE_BOOT_EN</td>
<td>EFUSE_BLK0</td>
<td>116</td>
<td>1</td>
</tr>
<tr>
<td>133</td>
<td>SECURE_BOOT_AGGRESSIVE_REVOKE</td>
<td>EFUSE_BLK0</td>
<td>117</td>
<td>1</td>
</tr>
<tr>
<td>134</td>
<td>FLASH_TPUW</td>
<td>EFUSE_BLK0</td>
<td>124</td>
<td>4</td>
</tr>
<tr>
<td>135</td>
<td>DIS_DOWNLOAD_MODE</td>
<td>EFUSE_BLK0</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>136</td>
<td>DIS_LEGACY_SPI_BOOT</td>
<td>EFUSE_BLK0</td>
<td>129</td>
<td>1</td>
</tr>
<tr>
<td>137</td>
<td>UART_PRINT_CHANNEL</td>
<td>EFUSE_BLK0</td>
<td>130</td>
<td>1</td>
</tr>
<tr>
<td>138</td>
<td>DIS_USB_DOWNLOAD_MODE</td>
<td>EFUSE_BLK0</td>
<td>132</td>
<td>1</td>
</tr>
<tr>
<td>139</td>
<td>ENABLE_SECURITY_DOWNLOAD</td>
<td>EFUSE_BLK0</td>
<td>133</td>
<td>1</td>
</tr>
<tr>
<td>140</td>
<td>UART_PRINT_CONTROL</td>
<td>EFUSE_BLK0</td>
<td>134</td>
<td>2</td>
</tr>
<tr>
<td>141</td>
<td>PIN_POWER_SELECTION</td>
<td>EFUSE_BLK0</td>
<td>136</td>
<td>1</td>
</tr>
<tr>
<td>142</td>
<td>FLASH_TYPE</td>
<td>EFUSE_BLK0</td>
<td>137</td>
<td>1</td>
</tr>
<tr>
<td>143</td>
<td>FORCE_SEND_RESUME</td>
<td>EFUSE_BLK0</td>
<td>138</td>
<td>1</td>
</tr>
<tr>
<td>144</td>
<td>SECURE_VERSION</td>
<td>EFUSE_BLK0</td>
<td>139</td>
<td>16</td>
</tr>
<tr>
<td>145</td>
<td>DISABLE_WAFER_VERSION_MAJOR</td>
<td>EFUSE_BLK0</td>
<td>160</td>
<td>1</td>
</tr>
<tr>
<td>146</td>
<td>DISABLE_BLK_VERSION_MAJOR</td>
<td>EFUSE_BLK0</td>
<td>161</td>
<td>1</td>
</tr>
<tr>
<td>147</td>
<td>MAC</td>
<td>EFUSE_BLK1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>148</td>
<td>MAC</td>
<td>EFUSE_BLK1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Line</td>
<td>Field</td>
<td>Block</td>
<td>Offset</td>
<td>Size</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------</td>
<td>---------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>149</td>
<td>MAC</td>
<td>EFUSE_BLK1</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>150</td>
<td>MAC</td>
<td>EFUSE_BLK1</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>151</td>
<td>MAC</td>
<td>EFUSE_BLK1</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>152</td>
<td>MAC</td>
<td>EFUSE_BLK1</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>153</td>
<td>SPI_PAD_CONFIG_CLK</td>
<td>EFUSE_BLK1</td>
<td>48</td>
<td>6</td>
</tr>
<tr>
<td>154</td>
<td>SPI_PAD_CONFIG_Q</td>
<td>EFUSE_BLK1</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>155</td>
<td>SPI_PAD_CONFIG_D</td>
<td>EFUSE_BLK1</td>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>156</td>
<td>SPI_PAD_CONFIG_CS</td>
<td>EFUSE_BLK1</td>
<td>66</td>
<td>6</td>
</tr>
<tr>
<td>157</td>
<td>SPI_PAD_CONFIG_HD</td>
<td>EFUSE_BLK1</td>
<td>72</td>
<td>6</td>
</tr>
<tr>
<td>158</td>
<td>SPI_PAD_CONFIG_WP</td>
<td>EFUSE_BLK1</td>
<td>78</td>
<td>6</td>
</tr>
<tr>
<td>159</td>
<td>SPI_PAD_CONFIG_DQS</td>
<td>EFUSE_BLK1</td>
<td>84</td>
<td>6</td>
</tr>
<tr>
<td>160</td>
<td>SPI_PAD_CONFIG_D4</td>
<td>EFUSE_BLK1</td>
<td>90</td>
<td>6</td>
</tr>
<tr>
<td>161</td>
<td>SPI_PAD_CONFIG_D5</td>
<td>EFUSE_BLK1</td>
<td>96</td>
<td>6</td>
</tr>
<tr>
<td>162</td>
<td>SPI_PAD_CONFIG_D6</td>
<td>EFUSE_BLK1</td>
<td>102</td>
<td>6</td>
</tr>
<tr>
<td>163</td>
<td>SPI_PAD_CONFIG_D7</td>
<td>EFUSE_BLK1</td>
<td>108</td>
<td>6</td>
</tr>
<tr>
<td>164</td>
<td>WAFER_VERSION_MAJOR</td>
<td>EFUSE_BLK1</td>
<td>114</td>
<td>2</td>
</tr>
<tr>
<td>165</td>
<td>WAFER_VERSION_MINOR_HI</td>
<td>EFUSE_BLK1</td>
<td>116</td>
<td>1</td>
</tr>
<tr>
<td>166</td>
<td>FLASH_VERSION</td>
<td>EFUSE_BLK1</td>
<td>117</td>
<td>4</td>
</tr>
<tr>
<td>167</td>
<td>BLK_VERSION_MAJOR</td>
<td>EFUSE_BLK1</td>
<td>121</td>
<td>2</td>
</tr>
<tr>
<td>168</td>
<td>PSRAM_VERSION</td>
<td>EFUSE_BLK1</td>
<td>124</td>
<td>4</td>
</tr>
<tr>
<td>169</td>
<td>PKG_VERSION</td>
<td>EFUSE_BLK1</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>170</td>
<td>WAFER_VERSION_MINOR_LO</td>
<td>EFUSE_BLK1</td>
<td>132</td>
<td>3</td>
</tr>
<tr>
<td>171</td>
<td>SYS_DATA_PART2</td>
<td>EFUSE_BLK10</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>172</td>
<td>OPTIONAL_UNIQUE_ID</td>
<td>EFUSE_BLK2</td>
<td>0</td>
<td>128</td>
</tr>
<tr>
<td>173</td>
<td>ADC_CALIB</td>
<td>EFUSE_BLK2</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>174</td>
<td>BLK_VERSION_MINOR</td>
<td>EFUSE_BLK2</td>
<td>132</td>
<td>3</td>
</tr>
<tr>
<td>175</td>
<td>TEMP_CALIB</td>
<td>EFUSE_BLK2</td>
<td>135</td>
<td>9</td>
</tr>
<tr>
<td>176</td>
<td>RTCCALIB_VIIDX_A10H</td>
<td>EFUSE_BLK2</td>
<td>144</td>
<td>8</td>
</tr>
<tr>
<td>177</td>
<td>RTCCALIB_VIIDX_A11H</td>
<td>EFUSE_BLK2</td>
<td>152</td>
<td>8</td>
</tr>
<tr>
<td>178</td>
<td>RTCCALIB_VIIDX_A12H</td>
<td>EFUSE_BLK2</td>
<td>160</td>
<td>8</td>
</tr>
<tr>
<td>179</td>
<td>RTCCALIB_VIIDX_A13H</td>
<td>EFUSE_BLK2</td>
<td>168</td>
<td>8</td>
</tr>
<tr>
<td>180</td>
<td>RTCCALIB_VIIDX_A20H</td>
<td>EFUSE_BLK2</td>
<td>176</td>
<td>8</td>
</tr>
<tr>
<td>181</td>
<td>RTCCALIB_VIIDX_A21H</td>
<td>EFUSE_BLK2</td>
<td>184</td>
<td>8</td>
</tr>
<tr>
<td>182</td>
<td>RTCCALIB_VIIDX_A22H</td>
<td>EFUSE_BLK2</td>
<td>192</td>
<td>8</td>
</tr>
<tr>
<td>183</td>
<td>RTCCALIB_VIIDX_A23H</td>
<td>EFUSE_BLK2</td>
<td>200</td>
<td>8</td>
</tr>
<tr>
<td>184</td>
<td>RTCCALIB_VIIDX_A10L</td>
<td>EFUSE_BLK2</td>
<td>208</td>
<td>6</td>
</tr>
<tr>
<td>185</td>
<td>RTCCALIB_VIIDX_A11L</td>
<td>EFUSE_BLK2</td>
<td>214</td>
<td>6</td>
</tr>
<tr>
<td>186</td>
<td>RTCCALIB_VIIDX_A12L</td>
<td>EFUSE_BLK2</td>
<td>220</td>
<td>6</td>
</tr>
<tr>
<td>187</td>
<td>RTCCALIB_VIIDX_A13L</td>
<td>EFUSE_BLK2</td>
<td>226</td>
<td>6</td>
</tr>
<tr>
<td>188</td>
<td>RTCCALIB_VIIDX_A20L</td>
<td>EFUSE_BLK2</td>
<td>232</td>
<td>6</td>
</tr>
<tr>
<td>189</td>
<td>RTCCALIB_VIIDX_A21L</td>
<td>EFUSE_BLK2</td>
<td>238</td>
<td>6</td>
</tr>
<tr>
<td>190</td>
<td>RTCCALIB_VIIDX_A22L</td>
<td>EFUSE_BLK2</td>
<td>244</td>
<td>6</td>
</tr>
<tr>
<td>191</td>
<td>RTCCALIB_VIIDX_A23L</td>
<td>EFUSE_BLK2</td>
<td>250</td>
<td>6</td>
</tr>
<tr>
<td>192</td>
<td>USER_DATA</td>
<td>EFUSE_BLK3</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>193</td>
<td>USER_DATA.MAC_CUSTOM</td>
<td>EFUSE_BLK3</td>
<td>200</td>
<td>48</td>
</tr>
<tr>
<td>194</td>
<td>KEY0</td>
<td>EFUSE_BLK4</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>195</td>
<td>KEY1</td>
<td>EFUSE_BLK5</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>196</td>
<td>KEY2</td>
<td>EFUSE_BLK6</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>197</td>
<td>KEY3</td>
<td>EFUSE_BLK7</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>198</td>
<td>KEY4</td>
<td>EFUSE_BLK8</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>199</td>
<td>KEY5</td>
<td>EFUSE_BLK9</td>
<td>0</td>
<td>256</td>
</tr>
</tbody>
</table>

Used bits in efuse table:

EFUSE_BLK0
[0 31] [0 0] [2 2] ... [32 38] [40 47] [49 51] [56 60] [68 70] [80 111] [116 117]...
[124 130] [132 154] [160 161]

EFUSE_BLK1
[0 122] [124 134]

EFUSE_BLK10
[0 255]

EFUSE_blk2

(continues on next page)
Chapter 2. API Reference

The number of bits not included in square brackets is free (some bits are reserved for Espressif). All fields are checked for overlapping.

To add fields to an existing field, use the Structured efuse fields technique. For example, adding the fields: SERIAL_NUMBER, MODEL_NUMBER and HARDWARE REV to an existing USER_DATA field. Use . (dot) to show an attachment in a field.

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.

You may get errors such as intersects with or out of range. Please see how to solve them in the Structured efuse fields article.

Bit Order

The eFuses bit order is little endian (see the example below), it means that eFuse bits are read and written from LSB to MSB:

```
$ espefuse.py dump

USER_DATA       (BLOCK3 ) [3 ] read_regs: 03020100 07060504 0B0A0908
  0F0E0D0C 13121111 17161514 1B1A1918 1F1E1D1C

BLOCK4         (BLOCK4 ) [4 ] read_regs: 03020100 07060504 0B0A0908
  0F0E0D0C 13121111 17161514 1B1A1918 1F1E1D1C

where is the register representation:

EFUSE_RD_USR_DATA0_REG = 0x03020100
EFUSE_RD_USR_DATA1_REG = 0x07060504
EFUSE_RD_USR_DATA2_REG = 0x0B0A0908
EFUSE_RD_USR_DATA3_REG = 0x0F0E0D0C
EFUSE_RD_USR_DATA4_REG = 0x13121111
EFUSE_RD_USR_DATA5_REG = 0x17161514
EFUSE_RD_USR_DATA6_REG = 0x1B1A1918
EFUSE_RD_USR_DATA7_REG = 0x1F1E1D1C

where is the byte representation:

(continues on next page)
For example, csv file describes the USER_DATA field, which occupies all 256 bits (a whole block).

<table>
<thead>
<tr>
<th>USER_DATA,</th>
<th>EFUSE_BLK3,</th>
<th>0, 256,</th>
<th>User data</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_DATA.FIELD1</td>
<td>EFUSE_BLK3,</td>
<td>16, 16,</td>
<td>Field1</td>
</tr>
</tbody>
</table>
| ID,              | EFUSE_BLK4, | 8, 3,   | ID bit[0..2]
| ,                | EFUSE_BLK4, | 16, 2,  | ID bit[3..4]
| ,                | EFUSE_BLK4, | 32, 3,  | ID bit[5..7]

Thus, reading the eFuse USER_DATA block written as above gives the following results:

```c
uint8_t buf[32] = { 0 };
esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, &buf, sizeof(buf) * 8);
// buf[0] = 0x00, buf[1] = 0x01, ... buf[31] = 0x1F

uint32_t field1 = 0;
size_t field1_size = ESP_EFUSE_USER_DATA[0]->bit_count; // can be used for this case because it only consists of one entry
esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, &field1, field1_size);
// field1 = 0x0302

uint32_t field1_1 = 0;
esp_efuse_read_field_blob(ESP_EFUSE_USER_DATA, &field1_1, 2); // reads only first 2 bits
// field1 = 0x0002

uint8_t id = 0;
size_t id_size = esp_efuse_get_field_size(ESP_EFUSE_ID); // returns 6
// size_t id_size = ESP_EFUSE_USER_DATA[0]->bit_count; // cannot be used because it consists of 3 entries. It returns 3 not 6.
esp_efuse_read_field_blob(ESP_EFUSE_ID, &id, id_size);
// id = 0x91
// b'100 10 001
// [3] [2] [3]

uint8_t id_1 = 0;
esp_efuse_read_field_blob(ESP_EFUSE_ID, &id_1, 3);
// id = 0x01
// b'001
```

Get eFuses During Build

There is a way to get the state of eFuses at the build stage of the project. There are two cmake functions for this:

- `espefuse_get_json_summary()` - It calls the `espefuse.py summary --format json` command and returns a json string (it is not stored in a file).
- `espefuse_get_efuse()` - It finds a given eFuse name in the json string and returns its property.

The json string has the following properties:

```json
{
    "MAC": {
        "bit_len": 48,
        "block": 0,
        "category": "identity",
        "description": "Factory MAC Address",
    }
}```
"efuse_type": "bytes:6",
"name": "MAC",
"pos": 0,
"readable": true,
"value": "94:b9:7e:5a:6e:58 (CRC 0xe2 OK)",
"word": 1,
"writeable": true
},

These functions can be used from a top-level project CMakeLists.txt (get-started/hello_world/CMakeLists.txt):

```c
# ...
project(hello_world)

espefuse_get_json_summary(efuse_json)
espefuse_get_efuse(ret_data \$\{efuse_json\} "MAC" "value")
message("MAC:" \$\{ret_data\})
```

The format of the `value` property is the same as shown in espefuse.py summary.

MAC:94:b9:7e:5a:6e:58 (CRC 0xe2 OK)

There is an example test system/efuse/CMakeLists.txt which adds a custom target efuse-summary. This allows you to run the idf.py efuse-summary command to read the required eFuses (specified in the efuse_names list) at any time, not just at project build time.

### Debug eFuse & Unit Tests

**Virtual eFuses**    The Kconfig option `CONFIG_EFUSE_VIRTUAL` virtualizes 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. During startup, the eFuses are copied to RAM. All eFuse operations (read and write) are performed with RAM instead of the real eFuse registers.

In addition to the `CONFIG_EFUSE_VIRTUAL` option there is `CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH` option that adds a feature to keep eFuses in flash memory. To use this mode the partition_table should have the `efuse` partition. partition.csv: "efuse_em, data, efuse, , 0x2000,". During startup, the eFuses are copied from flash or, in case if flash is empty, from real eFuse to RAM and then update flash. This option allows keeping eFuses after reboots (possible to test secure_boot and flash_encryption features with this option).

**Flash Encryption Testing**    Flash Encryption (FE) is a hardware feature that requires the physical burning of eFuses: key and FLASHCRYPT_CNT. If FE is not actually enabled then enabling the `CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH` option just gives testing possibilities and does not encrypt anything in the flash, even though the logs say encryption happens. The `bootloader_flash_write()` is adapted for this purpose. But if FE is already enabled on the chip and you run an application or bootloader created with the `CONFIG_EFUSE_VIRTUAL_KEEP_IN_FLASH` option then the flash encryption/decryption operations will work properly (data are encrypted as it is written into an encrypted flash partition and decrypted when they are read from an encrypted partition).

**espefuse.py**    esptool includes a useful tool for reading/writing ESP32-S2 eFuse bits - espefuse.py.

```
espefuse.py -p PORT summary
espefuse.py v4.6-dev
Connecting....
Detecting chip type... Unsupported detection protocol, switching and trying again...
```
Detecting chip type... ESP32-S2

--- Run "summary" command ---

**EFUSE_NAME (Block) Description** = [Meaningful Value] [Readable/Writeable] (Hex Value)

---

**Config fuses:**

- **WR_DIS (BLOCK0)**
  - disable programming of individual eFuses = 0 R/W (0x00000000)
  - disable reading from BLOCK4-10 = 0 R/W (0b0000000)

- **DIS_ICACHE (BLOCK0)**
  - set this bit to disable Icache = False R/W (0b0)

- **DIS_DCACHE (BLOCK0)**
  - set this bit to disable Dcache = False R/W (0b0)

- **DIS_TWAI (BLOCK0)**
  - set this bit to disable the TWAI Controller function = False R/W (0b0)

- **DIS_BOOT_REMAP (BLOCK0)**
  - disables capability to Remap RAM to ROM address space = False R/W (0b0)

- **DIS_LEGACY_SPI_BOOT (BLOCK0)**
  - set this bit to disable Legacy SPI boot mode = False R/W (0b0)

- **UART_PRINT_CHANNEL (BLOCK0)**
  - selects the default UART for printing boot message = UART0 R/W (0b0)

- **UART_PRINT_CONTROL (BLOCK0)**
  - set the default UART boot message output mode = Enable R/W (0b00)

- **PIN_POWER_SELECTION (BLOCK0)**
  - set default power supply for SPI flash is initialized when SPI flash is initialized = VDD3P3_CPU R/W (0b0)

- **BLOCK_USR_DATA (BLOCK3)**
  - user data = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...

- **BLOCK_SYS_DATA2 (BLOCK10)**
  - system data part 2 (reserved) = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...

**Flash fuses:**

- **FLASH_TPUW (BLOCK0)**
  - configures flash startup delay after SoC power-up; = 0 R/W (0x0)
  - value is 15; delay is 7.5 ms in unit of (ms/2). When the SPI flash type = 4 data lines R/W (0b0)

- **FORCE_SEND_RESUME (BLOCK0)**
  - an SPI flash resume = False R/W (0b0)

- **FLASH_VERSION (BLOCK1)**
  - flash version = 2 R/W (0x2)

**Identity fuses:**

- **BLOCK0_VERSION (BLOCK0)**
  - BLOCK0 efuse version = 0 R/W (0b00)

- **DISABLE_WAFER_VERSION_MAJOR (BLOCK0)**
  - disables check of wafer version major = False R/W (0b0)

- **DISABLE_BLK_VERSION_MAJOR (BLOCK0)**
  - disables check of blk version major = False R/W (0b0)

- **WAFER_VERSION_MAJOR (BLOCK1)**
  - WAFER_VERSION_MAJOR = 1 R/W (0b01)

(continues on next page)
### Chapter 2. API Reference

(continued from previous page)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAFER_VERSION_MINOR_HI (BLOCK1)</td>
<td>WAFER_VERSION_MINOR most significant bit</td>
<td>False</td>
</tr>
<tr>
<td>BLK_VERSION_MAJOR (BLOCK1)</td>
<td>BLK_VERSION_MAJOR</td>
<td></td>
</tr>
<tr>
<td>PSRAM_VERSION (BLOCK1)</td>
<td>PSRAM version</td>
<td></td>
</tr>
<tr>
<td>PKG_VERSION (BLOCK1)</td>
<td>Package version</td>
<td></td>
</tr>
<tr>
<td>WAFER_VERSION_MINOR_LO (BLOCK1)</td>
<td>WAFER_VERSION_MINOR least significant bit</td>
<td></td>
</tr>
<tr>
<td>OPTIONAL_UNIQUE_ID (BLOCK2)</td>
<td>Optional unique 128-bit ID</td>
<td>ea 0e c6 f1 01 f2 38 82 e9 98 5b 59 81 fe 00 02</td>
</tr>
<tr>
<td>BLK_VERSION_MINOR (BLOCK2)</td>
<td>BLK_VERSION_MINOR of BLOCK2</td>
<td>ADC calib V2</td>
</tr>
<tr>
<td>WAFER_VERSION_MINOR (BLOCK)</td>
<td>calc WAFER VERSION MINOR</td>
<td></td>
</tr>
<tr>
<td>OPTIONAL_UNIQUE_ID (BLOCK1)</td>
<td>Optional unique 128-bit ID</td>
<td>ea 0e c6 f1 01 f2 38 82 e9 98 5b 59 81 fe 00 02</td>
</tr>
<tr>
<td>BLK_VERSION_MINOR (BLOCK0)</td>
<td>BLK_VERSION_MINOR</td>
<td>0</td>
</tr>
<tr>
<td>WAFER_VERSION_MINOR_HI = 0 R/W (0x0)</td>
<td>&lt;&lt; 3 + WAFER_VERSION_MINOR_LO</td>
<td></td>
</tr>
</tbody>
</table>

(--read only)

Jtag fuses:
- SOFT_DIS_JTAG (BLOCK0) Software disables JTAG. When disabled; JT = False R/W (0b0)
- HARD_DIS_JTAG (BLOCK0) Hardware disables JTAG permanently

Mac fuses:
- MAC (BLOCK1) MAC address
- CUSTOM_MAC (BLOCK3) Custom MAC

Security fuses:
- DIS_DOWNLOAD_ICACHE (BLOCK0) Disables Icache when SoC is in download mode
- DIS_DOWNLOAD_DCACHE (BLOCK0) Disables Dcache when SoC is in download mode
- DIS_FORCE_DOWNLOAD (BLOCK0) Set this bit to disable the hip into download mode
- SPI_BOOT_CRYPT_CNT (BLOCK0) Enables flash encryption when 1 or 3 bits are set and disabled otherwise
- SECURE_BOOT_KEY_REV0E0 (BLOCK0) Revoke 1st secure boot key
- SECURE_BOOT_KEY_REV0E1 (BLOCK0) Revoke 2nd secure boot key
- SECURE_BOOT_KEY_REV0E2 (BLOCK0) Revoke 3rd secure boot key
- KEY_PURPOSE_0 (BLOCK0) Purpose of KEY0
- KEY_PURPOSE_1 (BLOCK0) Purpose of KEY1
- KEY_PURPOSE_2 (BLOCK0) Purpose of KEY2
- KEY_PURPOSE_3 (BLOCK0) Purpose of KEY3

(continues on next page)
Chapter 2. API Reference

(continued from previous page)

KEY_PURPOSE_4 (BLOCK0) Purpose of KEY4 .....
-> USER R/W (0x0)

KEY_PURPOSE_5 (BLOCK0) Purpose of KEY5 .....
-> USER R/W (0x0)

SECURE_BOOT_EN (BLOCK0) Set this bit to enable secure...
-> boot = False R/W (0b0)

SECURE_BOOT_AGGRESSIVE_REVOKE (BLOCK0) Set this bit to enable...
-> aggressive secure boot key = False R/W (0b0)

DIS_DOWNLOAD_MODE (BLOCK0) Set this bit to disable all...
-> download boot modes = False R/W (0b0)

ENABLE_SECURITY_DOWNLOAD (BLOCK0) Set this bit to enable secure...
-> UART download mode ( = False R/W (0b0)

read/write flash only)

SECURE_VERSION (BLOCK0) Secure version (used by ESP-IDF...
-> anti-rollback feat = 0 R/W (0x0000)

BLOCK_KEY0 (BLOCK4) Purpose: USER

Key0 or user data
- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
- 00 00 00 00 00 00 R/W

BLOCK_KEY1 (BLOCK5) Purpose: USER

Key1 or user data
- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
- 00 00 00 00 00 00 R/W

BLOCK_KEY2 (BLOCK6) Purpose: USER

Key2 or user data
- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
- 00 00 00 00 00 00 R/W

BLOCK_KEY3 (BLOCK7) Purpose: USER

Key3 or user data
- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
- 00 00 00 00 00 00 R/W

BLOCK_KEY4 (BLOCK8) Purpose: USER

Key4 or user data
- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
- 00 00 00 00 00 00 R/W

BLOCK_KEY5 (BLOCK9) Purpose: USER

Key5 or user data
- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
- 00 00 00 00 00 00 R/W

Spi Pad fuses:

SPI_PAD_CONFIG_CLK (BLOCK1) SPI_PAD_configure CLK  .....
-> 0 R/W (0b000000)

SPI_PAD_CONFIG_Q (BLOCK1) SPI_PAD_configure Q(D1)  .....
-> 0 R/W (0b000000)

SPI_PAD_CONFIG_D (BLOCK1) SPI_PAD_configure D(D0)  .....
-> 0 R/W (0b000000)

SPI_PAD_CONFIG_CS (BLOCK1) SPI_PAD_configure CS  .....
-> 0 R/W (0b000000)

SPI_PAD_CONFIG_HD (BLOCK1) SPI_PAD_configure HD(D3)  .....
-> 0 R/W (0b000000)

SPI_PAD_CONFIG_WP (BLOCK1) SPI_PAD_configure WP(D2)  .....
-> 0 R/W (0b000000)

(continues on next page)
<table>
<thead>
<tr>
<th>SPI_PAD_CONFIG_DQS (BLOCK1)</th>
<th>SPI_PAD_configure DQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0 R/W (0b000000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPI_PAD_CONFIG_D4 (BLOCK1)</th>
<th>SPI_PAD_configure D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0 R/W (0b000000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPI_PAD_CONFIG_D5 (BLOCK1)</th>
<th>SPI_PAD_configure D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0 R/W (0b000000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPI_PAD_CONFIG_D6 (BLOCK1)</th>
<th>SPI_PAD_configure D6</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0 R/W (0b000000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPI_PAD_CONFIG_D7 (BLOCK1)</th>
<th>SPI_PAD_configure D7</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0 R/W (0b000000)</td>
<td></td>
</tr>
</tbody>
</table>

### Usb fuses:

<table>
<thead>
<tr>
<th>DIS_USB (BLOCK0)</th>
<th>Set this bit to disable USB OTG...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ function = False R/W (0b0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USB_EXCHG_PINS (BLOCK0)</th>
<th>Set this bit to exchange USB D+...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ and D- pins = False R/W (0b0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USB_EXT_PHY_ENABLE (BLOCK0)</th>
<th>Set this bit to enable external...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ USB PHY = False R/W (0b0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USB_FORCE_NOPERSIST (BLOCK0)</th>
<th>If set; forces USB BVALID to 1...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ = False R/W (0b0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DIS_USB_DOWNLOAD_MODE (BLOCK0)</th>
<th>Set this bit to disable use of...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ USB OTG in UART dow = False R/W (0b0)</td>
<td>nload boot mode</td>
</tr>
</tbody>
</table>

### Vdd fuses:

<table>
<thead>
<tr>
<th>VDD_SPI_XPD (BLOCK0)</th>
<th>If VDD_SPI_FORCE is 1; this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ value determines if th = False R/W (0b0)</td>
<td>e VDD_SPI regulator is powered...</td>
</tr>
<tr>
<td>→ on</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VDD_SPI_TIEH (BLOCK0)</th>
<th>Determines VDD_SPI voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ VDD_SPI connects to 1.8 V LDO R/W (0b0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VDD_SPI_FORCE (BLOCK0)</th>
<th>Set this bit to use XPD_VDD_PSI...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ REG and VDD_SPI_TI = False R/W (0b0)</td>
<td>EH to configure VDD_SPI LDO</td>
</tr>
</tbody>
</table>

### Wdt fuses:

<table>
<thead>
<tr>
<th>WDT_DELAY_SEL (BLOCK0)</th>
<th>RTC watchdog timeout threshold;...</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ in unit of slow cl = 40000 R/W (0b00)</td>
<td>ock cycle</td>
</tr>
</tbody>
</table>

Flash voltage (VDD_SPI) determined by GPIO45 on reset (GPIO45=High: VDD_SPI pin is... |
| → powered from internal 1.8V LDO |

GPIO45=Low or NC: VDD_SPI pin is powered directly from VDD3P3_RTC_IO via resistor... |
| → Rspl. Typically this voltage is 3.3 V). |

To get a dump for all eFuse registers.

```bash
esefuse.py -p PORT dump
```

```bash
esefuse.py v4.6-dev
```

Connecting....

Detecting chip type... Unsupported detection protocol, switching and trying again... |

Detecting chip type... ESP32-S2

```bash
BLOCK0
  { [0 ] read_regs: 00000000 00000000 00000000 ...}
MAC_SPI_8M_0 (BLOCK1)
  { [1 ] read_regs: 79b3b954 000058cf 00000000 ...}
10440000 00000000 00000000
BLOCK_SYS_DATA (BLOCK2)
  } [2 ] read_regs: f1c60eea 8238f201 595b98e9 ...0200fe81 1c549f24 88491102 06461421 070c2083
```
### Header File

- components/efuse/esp32s2/include/esp_efuse_chip.h
  - This header file can be included with:
    ```c
    #include "esp_efuse_chip.h"
    ```
  - This header file is a part of the API provided by the efuse component. To declare that your component depends on efuse, add the following to your CMakeLists.txt:
    ```cmake
    REQUIRES efuse
    ```
    or
    ```cmake
    PRIV_REQUIRES efuse
    ```

### Enumerations

**enum** `esp_efuse_block_t`

Type of eFuse blocks ESP32S2.

**Values:**

- **enumerator** `EFUSE_BLK0`
  - Number of eFuse BLOCK0. REPEAT_DATA
- **enumerator** `EFUSE_BLK1`
  - Number of eFuse BLOCK1. MAC_SPI_8M_SYS
- **enumerator** `EFUSE_BLK2`
  - Number of eFuse BLOCK2. SYS_DATA_PART1
- **enumerator** `EFUSE_BLK_SYS_DATA_PART1`
Chapter 2. API Reference

Number of eFuse BLOCK2. SYS_DATA_PART1

enumerator **EFUSE_BLK3**
Number of eFuse BLOCK3. USER_DATA

enumerator **EFUSE_BLK_USER_DATA**
Number of eFuse BLOCK3. USER_DATA

enumerator **EFUSE_BLK4**
Number of eFuse BLOCK4. KEY0

enumerator **EFUSE_BLK_KEY0**
Number of eFuse BLOCK4. KEY0

enumerator **EFUSE_BLK5**
Number of eFuse BLOCK5. KEY1

enumerator **EFUSE_BLK_KEY1**
Number of eFuse BLOCK5. KEY1

enumerator **EFUSE_BLK6**
Number of eFuse BLOCK6. KEY2

enumerator **EFUSE_BLK_KEY2**
Number of eFuse BLOCK6. KEY2

enumerator **EFUSE_BLK7**
Number of eFuse BLOCK7. KEY3

enumerator **EFUSE_BLK_KEY3**
Number of eFuse BLOCK7. KEY3

enumerator **EFUSE_BLK8**
Number of eFuse BLOCK8. KEY4

enumerator **EFUSE_BLK_KEY4**
Number of eFuse BLOCK8. KEY4

enumerator **EFUSE_BLK9**
Number of eFuse BLOCK9. KEY5

enumerator **EFUSE_BLK_KEY5**
Number of eFuse BLOCK9. KEY5

enumerator **EFUSE_BLK_KEY_MAX**

enumerator **EFUSE_BLK10**
Number of eFuse BLOCK10. SYS_DATA_PART2
enumerator **EFUSE_BLK_SYS_DATA_PART2**

Number of eFuse BLOCK10. SYS_DATA_PART2

enumerator **EFUSE_BLK_MAX**

enum **esp_efuse_coding_scheme_t**

Type of coding scheme.

_values:_

enumerator **EFUSE_CODING_SCHEME_NONE**

None

enumerator **EFUSE_CODING_SCHEME_RS**

Reed-Solomon coding

enum **esp_efuse_purpose_t**

Type of key purpose.

_values:_

enumerator **ESP_EFUSE_KEY_PURPOSE_USER**

User purposes (software-only use)

enumerator **ESP_EFUSE_KEY_PURPOSE_RESERVED**

Reserved

enumerator **ESP_EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_1**

XTS_AES_256_KEY_1 (flash/PSRAM encryption)

enumerator **ESP_EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_2**

XTS_AES_256_KEY_2 (flash/PSRAM encryption)

enumerator **ESP_EFUSE_KEY_PURPOSE_XTS_AES_128_KEY**

XTS_AES_128_KEY (flash/PSRAM encryption)

enumerator **ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL**

HMAC Downstream mode

enumerator **ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG**

JTAG soft enable key (uses HMAC Downstream mode)

enumerator **ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE**

Digital Signature peripheral key (uses HMAC Downstream mode)

enumerator **ESP_EFUSE_KEY_PURPOSE_HMAC_UP**

HMAC Upstream mode

enumerator **ESP_EFUSE_KEY_PURPOSE_SECURE_BOOT_DIGEST0**

SECURE_BOOT_DIGEST0 (Secure Boot key digest)
enumerator **ESP_EFUSE_KEY_PURPOSE_SECURE_BOOT_DIGEST1**
SECURE_BOOT_DIGEST1 (Secure Boot key digest)

enumerator **ESP_EFUSE_KEY_PURPOSE_SECURE_BOOT_DIGEST2**
SECURE_BOOT_DIGEST2 (Secure Boot key digest)

enumerator **ESP_EFUSE_KEY_PURPOSE_MAX**
MAX PURPOSE

**Header File**

- components/efuse/include/esp_efuse.h
- This header file can be included with:

```
#include "esp_efuse.h"
```
- This header file is a part of the API provided by the efuse component. To declare that your component depends on efuse, add the following to your CMakeLists.txt:

```
REQUIRES efuse
```
or
```
PRIV_REQUIRES efuse
```

**Functions**

```c
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.

**Note:** Please note that reading in the batch mode does not show uncommitted changes.

**Parameters**

- **field** [in] A pointer to the structure describing the fields of efuse.
- **dst** [out] A pointer to array that will contain the result of reading.
- **dst_size_bits** [in] 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.

**Returns**

- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

```c
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.
Note: Please note that reading in the batch mode does not show uncommitted changes.

**Parameters**
- `field` [in] A pointer to the structure describing the fields of efuse.

**Returns**
- 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.

```c
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.

Note: Please note that reading in the batch mode does not show uncommitted changes.

**Parameters**
- `field` [in] A pointer to the structure describing the fields of efuse.
- `out_cnt` [out] A pointer that will contain the number of programmed as “1” bits.

**Returns**
- ESP_OK: The operation was successfully completed.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.

```c
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.

**Parameters**
- `field` [in] A pointer to the structure describing the fields of efuse.
- `src` [in] A pointer to array that contains the data for writing.
- `src_size_bits` [in] The number of bits required to write.

**Returns**
- ESP_OK: The operation was successfully completed.
- ESP_ERR.INVISIBLE_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.

```c
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.

**Parameters**
- `field` [in] A pointer to the structure describing the fields of efuse.
- `cnt` [in] Required number of programmed as “1” bits.

**Returns**
- 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.
Chapter 2.  API Reference

```c
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.

Parameters

field  [in] Pointer to the structure describing the efuse field.

Returns

• 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.
```

```c
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.

Parameters

blk  [in] Block number of eFuse. (EFUSE_BLK1, EFUSE_BLK2 and EFUSE_BLK3)

Returns

• 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.
```

```c
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.

Parameters

blk  [in] Block number of eFuse. (EFUSE_BLK1, EFUSE_BLK2 and EFUSE_BLK3)

Returns

• 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.
```

```c
int esp_efuse_get_field_size (const esp_efuse_desc_t *field[])

Returns the number of bits used by field.

Parameters

field  [in] A pointer to the structure describing the fields of efuse.

Returns

Returns the number of bits used by field.
```

```c
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)

Note: Please note that reading in the batch mode does not show uncommitted changes.
```

Parameters

• blk  [in] Block number of eFuse.
• num_reg  [in] The register number in the block.

Returns Value of register
**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)

**Parameters**
- **blk**  – [in] Block number of eFuse.
- **num_reg**  – [in] The register number in the block.
- **val**  – [in] Value to write.

**Returns**
- ESP_OK: The operation was successfully completed.
- ESP_ERR_EFUSE_REPEATED_PROG: Error repeated programming of programmed bits is strictly forbidden.

**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.

**Parameters** blk  – [in] Block number of eFuse.

**Returns** Return efuse coding scheme for blocks.

**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.

**Note:** Please note that reading in the batch mode does not show uncommitted changes.

**Parameters**
- **blk**  – [in] Block number of eFuse.
- **dst_key**  – [in] A pointer to array that will contain the result of reading.
- **size_bits**  – [in] The number of bits required to read.

**Returns**
- 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_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.

**Parameters**
- **blk**  – [in] Block number of eFuse.
- **src_key**  – [in] A pointer to array that contains the key for writing.
- **size_bits**  – [in] The number of bits required to write.

**Returns**
- 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.

**uint32_t** esp_efuse_get_pkg_ver (void)

Returns chip package from efuse.

**Returns** chip package
void esp_efuse_reset (void)
Reset efuse write registers.
Efuse write registers are written to zero, to negate any changes that have been staged here.

Note: This function is not threadsafe, if calling code updates efuse values from multiple tasks then this is caller’s responsibility to serialise.

esp_err_t esp_efuse_disable_rom_download_mode (void)
Disable ROM Download Mode via eFuse.
Permanently disables the ROM Download Mode feature. Once disabled, if the SoC is booted with strapping pins set for ROM Download Mode then an error is printed instead.

Note: Not all SoCs support this option. An error will be returned if called on an ESP32 with a silicon revision lower than 3, as these revisions do not support this option.

Note: If ROM Download Mode is already disabled, this function does nothing and returns success.

Returns
- ESP_OK If the eFuse was successfully burned, or had already been burned.
- ESP_ERR_NOT_SUPPORTED (ESP32 only) This SoC is not capable of disabling UART download mode
- ESP_ERR_INVALID_STATE (ESP32 only) This eFuse is write protected and cannot be written

esp_err_t esp_efuse_set_rom_log_scheme (esp_efuse_rom_log_scheme_t log_scheme)
Set boot ROM log scheme via eFuse.

Note: By default, the boot ROM will always print to console. This API can be called to set the log scheme only once per chip, once the value is changed from the default it can’t be changed again.

Parameters log_scheme – Supported ROM log scheme

Returns
- ESP_OK If the eFuse was successfully burned, or had already been burned.
- ESP_ERR_NOT_SUPPORTED (ESP32 only) This SoC is not capable of setting ROM log scheme
- ESP_ERR_INVALID_STATE This eFuse is write protected or has been burned already

esp_err_t esp_efuse_enable_rom_secure_download_mode (void)
Switch ROM Download Mode to Secure Download mode via eFuse.
Permanently enables Secure Download mode. This mode limits the use of ROM Download Mode functions to simple flash read, write and erase operations, plus a command to return a summary of currently enabled security features.

Note: If Secure Download mode is already enabled, this function does nothing and returns success.

Note: Disabling the ROM Download Mode also disables Secure Download Mode.
Returns
• ESP_OK If the eFuse was successfully burned, or had already been burned.
• ESP_ERR_INVALID_STATE ROM Download Mode has been disabled via eFuse, so secure Download mode is unavailable.

uint32_t esp_efuse_read_secure_version (void)

Returns Secure version from eFuse field.

Parameters

Returns

bool esp_efuse_check_secure_version (uint32_t secure_version)

Check secure version from app and secure_version and from eFuse field.

Parameters secure_version – Secure version from app.

Returns
• True: If version of app is equal or more than secure_version from eFuse.

esp_err_t esp_efuse_update_secure_version (uint32_t secure_version)

Write eFuse field by secure_version value.


Returns
• ESP_OK: Successful.
• ESP_FAIL: secure_version of app cannot be set to eFuse field.
• ESP_ERR_NOT_SUPPORTED: Antirollback is not supported with the 3/4 and Repeat coding scheme.

esp_err_t esp_efuse_batch_write_begin (void)

Set the batch mode of writing fields.

This mode allows you to write the fields in the batch mode when need to burn several eFuses at one time. To enable batch mode call begin() then perform as usually the necessary operations read and write and at the end call commit() to actually burn all written eFuses. The batch mode can be used nested. The commit will be done by the last commit() function. The number of begin() functions should be equal to the number of commit() functions.

Note: If batch mode is enabled by the first task, at this time the second task cannot write/read eFuses. The second task will wait for the first task to complete the batch operation.

// Example of using the batch writing mode.

// set the batch writing mode
esp_efuse_batch_write_begin();

// use any writing functions as usual
esp_efuse_write_field_blob(ESP_EFUSE_...);
esp_efuse_write_field_cnt(ESP_EFUSE_...);
esp_efuse_set_write_protect(EFUSE_BLKx);
esp_efuse_write_reg(EFUSE_BLKx, ...);
esp_efuse_write_block(EFUSE_BLKx, ...);
esp_efuse_write(ESP_EFUSE_1, 3); // ESP_EFUSE_1 == 1, here we write a new...
value = 3. The changes will be burn by the commit() function.
esp_efuse_read...(ESP_EFUSE_1); // this function returns ESP_EFUSE_1 == 1...
because uncommitted changes are not readable, it will be available only...
after commit.

(continues on next page)
// esp_efuse_batch_write APIs can be called recursively.
esp_efuse_batch_write_begin();
esp_efuse_set_write_protect(EFUSE_BLKx);
esp_efuse_batch_write_commit(); // the burn will be skipped here, it will be...
...done in the last commit();
...

// Write all of these fields to the efuse registers
esp_efuse_batch_write_commit();
esp_efuse_read...(ESP_EFUSE_1); // this function returns ESP_EFUSE_1 == 3.

Note: Please note that reading in the batch mode does not show uncommitted changes.

Returns
- ESP_OK: Successful.

`esp_err_t esp_efuse_batch_write_cancel (void)`
Reset the batch mode of writing fields.
It will reset the batch writing mode and any written changes.

Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_STATE: Tha batch mode was not set.

`esp_err_t esp_efuse_batch_write_commit (void)`
Writes all prepared data for the batch mode.
Must be called to ensure changes are written to the efuse registers. After this the batch writing mode will be reset.

Returns
- ESP_OK: Successful.
- ESP_ERR_INVALID_STATE: The deferred writing mode was not set.

`bool esp_efuse_block_is_empty (esp_efuse_block_t block)`
Checks that the given block is empty.

Returns
- True: The block is empty.
- False: The block is not empty or was an error.

`bool esp_efuse_get_key_dis_read (esp_efuse_block_t block)`
Returns a read protection for the key block.

Parameters
- block [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns
- True: The key block is read protected
- False: The key block is readable.

`esp_err_t esp_efuse_set_key_dis_read (esp_efuse_block_t block)`
Sets a read protection for the key block.

Parameters
- block [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns
- ESP_OK: Successful.
- 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.
bool esp_efuse_get_key_dis_write (esp_efuse_block_t block)

Returns a write protection for the key block.

Parameters block [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns True: The key block is write protected False: The key block is writeable.

esp_err_t esp_efuse_set_key_dis_write (esp_efuse_block_t block)

Sets a write protection for the key block.

Parameters block [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns
• ESP_OK: Successful.
• 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.

bool esp_efuse_key_block_unused (esp_efuse_block_t block)

Returns true if the key block is unused, false otherwise.

An unused key block is all zero content, not read or write protected, and has purpose 0 (ESP_EFUSE_KEY_PURPOSE_USER)

Parameters block –key block to check.

Returns
• True if key block is unused,
• False if key block is used or the specified block index is not a key block.

bool esp_efuse_find_purpose (esp_efuse_purpose_t purpose, esp_efuse_block_t *block)

Find a key block with the particular purpose set.

Parameters
• purpose –[in] Purpose to search for.
• block –[out] Pointer in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX which will be set to the key block if found. Can be NULL, if only need to test the key block exists.

Returns
• True: If found,
• False: If not found (value at block pointer is unchanged).

bool esp_efuse_get_keypurpose_dis_write (esp_efuse_block_t block)

Returns a write protection of the key purpose field for an efuse key block.

Note: For ESP32: no key purpose, it returns always True.

Parameters block [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns True: The key purpose is write protected False: The key purpose is writeable.

esp_efuse_purpose_t esp_efuse_get_key_purpose (esp_efuse_block_t block)

Returns the current purpose set for an efuse key block.

Parameters block [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns
• Value: If Successful, it returns the value of the purpose related to the given key block.
• ESP_EFUSE_KEY_PURPOSE_MAX: Otherwise.

cost esp_efuse_desc_t **esp_efuse_get_purpose_field (esp_efuse_block_t block)

Returns a pointer to a key purpose for an efuse key block.

To get the value of this field use esp_efuse_read_field_blob() or esp_efuse_get_key_purpose().
Parameters `block`  - [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns Pointer: If Successful returns a pointer to the corresponding efuse field otherwise NULL.

```c
const esp_efuse_desc_t **esp_efuse_get_key (esp_efuse_block_t block)
```

Returns a pointer to a key block.

Parameters `block` - [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

```c
void esp_efuse_set_key_purpose (esp_efuse_block_t block, esp_efuse_purpose_t purpose)
```

Sets a key purpose for an efuse key block.

Parameters
- `block` - [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX
- `purpose` - [in] Key purpose.

Returns
- ESP_OK: Successful.
- 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.

```c
esp_err_t esp_efuse_set_keypurpose_dis_write (esp_efuse_block_t block)
```

Sets a write protection of the key purpose field for an efuse key block.

Parameters `block` - [in] A key block in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX

Returns
- ESP_OK: Successful.
- 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.

```c
esp_efuse_block_t esp_efuse_find_unused_key_block (void)
```

Search for an unused key block and return the first one found.

See esp_efuse_key_block_unused for a description of an unused key block.

Returns First unused key block, or EFUSE_BLK_KEY_MAX if no unused key block is found.

```c
unsigned esp_efuse_count_unused_key_blocks (void)
```

Return the number of unused efuse key blocks in the range EFUSE_BLK_KEY0..EFUSE_BLK_KEY_MAX.

`bool esp_efuse_get_digest_revoke (unsigned num_digest)`

Returns the status of the Secure Boot public key digest revocation bit.

Parameters `num_digest` - [in] The number of digest in range 0..2

Returns
- True: If key digest is revoked,
- False: If key digest is not revoked.

```c
esp_err_t esp_efuse_set_digest_revoke (unsigned num_digest)
```

Sets the Secure Boot public key digest revocation bit.

Parameters `num_digest` - [in] The number of digest in range 0..2

Returns
- ESP_OK: Successful.
- 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.
**bool esp_efuse_get_write_protect_of_digest_revoke** *(unsigned num_digest)*

Returns a write protection of the Secure Boot public key digest revocation bit.

**Parameters**
- num_digest - [in] The number of digest in range 0..2

**Returns**
- True: The revocation bit is write protected.
- False: The revocation bit is writeable.

**esp_err_t esp_efuse_set_write_protect_of_digest_revoke** *(unsigned num_digest)*

Sets a write protection of the Secure Boot public key digest revocation bit.

**Parameters**
- num_digest - [in] The number of digest in range 0..2

**Returns**

- ESP_OK: Successful.
- 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.

**esp_err_t esp_efuse_write_key** *(esp_efuse_block_t block, esp_efuse_purpose_t purpose, const void* key, size_t key_size_bytes)*

Program a block of key data to an efuse block.

The burn of a key, protection bits, and a purpose happens in batch mode.

**Parameters**

- block - [in] Block to read purpose for. Must be in range EFUSE_BLK_KEY0 to EFUSE_BLK_KEY_MAX. Key block must be unused (esp_efuse_key_block_unused).
- purpose - [in] Purpose to set for this key. Purpose must be already unset.
- key - [in] Pointer to data to write.
- key_size_bytes - [in] Bytes length of data to write.

**Returns**

- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_INVALID_STATE: Error in efuses state, unused block not found.
- 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.

**esp_err_t esp_efuse_write_keys** *(const esp_efuse_purpose_t purposes[], uint8_t keys[][32], unsigned number_of_keys)*

Program keys to unused efuse blocks.

The burn of keys, protection bits, and purposes happens in batch mode.

**Parameters**

- purposes - [in] Array of purposes (purpose[number_of_keys]).
- keys - [in] Array of keys (uint8_t keys[number_of_keys][32]). Each key is 32 bytes long.
- number_of_keys - [in] The number of keys to write (up to 6 keys).

**Returns**

- ESP_OK: Successful.
- ESP_ERR_INVALID_ARG: Error in the passed arguments.
- ESP_ERR_INVALID_STATE: Error in efuses state, unused block not found.
- ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS: Error not enough unused key blocks available
- 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.

**esp_err_t esp_secure_boot_read_key_digests** *(esp_secure_boot_key_digests_t *trusted_key_digests)*

Read key digests from efuse. Any revoked/missing digests will be marked as NULL.
Parameters trusted_key_digests –[out] Trusted keys digests, stored in this parameter after successfully completing this function. The number of digests depends on the SOC’s capabilities.

Returns

- ESP_OK: Successful.
- ESP_FAIL: If trusted_keys is NULL or there is no valid digest.

esp_err_t esp_efuse_check_errors (void)
Checks eFuse errors in BLOCK0.

It does a BLOCK0 check if eFuse EFUSE_ERR_RST_ENABLE is set. If BLOCK0 has an error, it prints the error and returns ESP_FAIL, which should be treated as esp_restart.

Note: Refers to ESP32-C3 only.

Returns

- ESP_OK: No errors in BLOCK0.
- ESP_FAIL: Error in BLOCK0 requiring reboot.

Structures

struct esp_efuse_desc_t
Type definition for an eFuse field.

Public Members

esp_efuse_block_t efuse_block
Block of eFuse

uint8_t bit_start
Start bit [0..255]

uint16_t bit_count
Length of bit field [1..-]

struct esp_secure_boot_key_digests_t
Pointers to the trusted key digests.
The number of digests depends on the SOC’s capabilities.

Public Members

const void *key_digests[3]
Pointers to the key digests

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.

**ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS**

Error not enough unused key blocks available

**ESP_ERR_DAMAGED_READING**

Error. Burn or reset was done during a reading operation leads to damage read data. This error is internal to the efuse component and not returned by any public API.

### Enumerations

enum `esp_efuse_rom_log_scheme_t`

Type definition for ROM log scheme.

*Values:*

- **ESP_EFUSE_ROM_LOG_ALWAYS_ON**
  Always enable ROM logging

- **ESP_EFUSE_ROM_LOG_ON_GPIO_LOW**
  ROM logging is enabled when specific GPIO level is low during start up

- **ESP_EFUSE_ROM_LOG_ON_GPIO_HIGH**
  ROM logging is enabled when specific GPIO level is high during start up

- **ESP_EFUSE_ROM_LOG_ALWAYS_OFF**
  Disable ROM logging permanently

### 2.9.8 Error Code 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 Codes Reference*.

### API Reference

**Header File**

- components/esp_common/include/esp_check.h
• This header file can be included with:

```c
#include "esp_check.h"
```

### Macros

**ESP_RETURN_ON_ERROR** (x, log_tag, format, ...)

Macro which can be used to check the error code. If the code is not ESP_OK, it prints the message and returns. In the future, we want to switch to C++20. We also want to become compatible with clang. Hence, we provide two versions of the following macros. The first one is using the GNU extension `##__VA_ARGS__`. The second one is using the C++20 feature `VA_OPT(...)`. This allows users to compile their code with standard C++20 enabled instead of the GNU extension. Below C++20, we haven’t found any good alternative to using `##__VA_ARGS__`. Macro which can be used to check the error code. If the code is not ESP_OK, it prints the message and returns.

**ESP_RETURN_ON_ERROR_ISR** (x, log_tag, format, ...)

A version of ESP_RETURN_ON_ERROR() macro that can be called from ISR.

**ESP_GOTO_ON_ERROR** (x, goto_tag, log_tag, format, ...)

Macro which can be used to check the error code. If the code is not ESP_OK, it prints the message, sets the local variable ‘ret’ to the code, and then exits by jumping to ‘goto_tag’.

**ESP_GOTO_ON_ERROR_ISR** (x, goto_tag, log_tag, format, ...)

A version of ESP_GOTO_ON_ERROR() macro that can be called from ISR.

**ESP_RETURN_ON_FALSE** (a, err_code, log_tag, format, ...)

Macro which can be used to check the condition. If the condition is not ‘true’, it prints the message and returns with the supplied ‘err_code’.

**ESP_RETURN_ON_FALSE_ISR** (a, err_code, log_tag, format, ...)

A version of ESP_RETURN_ON_FALSE() macro that can be called from ISR.

**ESP_GOTO_ON_FALSE** (a, err_code, goto_tag, log_tag, format, ...)

Macro which can be used to check the condition. If the condition is not ‘true’, it prints the message, sets the local variable ‘ret’ to the supplied ‘err_code’, and then exits by jumping to ‘goto_tag’.

**ESP_GOTO_ON_FALSE_ISR** (a, err_code, goto_tag, log_tag, format, ...)

A version of ESP_GOTO_ON_FALSE() macro that can be called from ISR.

### Header File

- `components/esp_common/include/esp_err.h`
- This header file can be included with:

```c
#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.

- **Parameters** code – esp_err_t error code
- **Returns** string error message
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.

Parameters
- code - esp_err_t error code
- buf [out] 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).

Returns buf containing the string error message

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_NOT_FINISHED
Operation has not fully completed

ESP_ERR_NOT_ALLOWED
Operation is not allowed

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_ERR_HW_CRYPTO_BASE
Starting number of HW cryptography module error codes

ESP_ERR_MEMPROT_BASE
Starting number of Memory Protection API 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 int esp_err_t

2.9.9 ESP HTTPS OTA

Overview

esp_https_ota provides simplified APIs to perform firmware upgrades over HTTPS. It is an abstraction layer over the existing OTA APIs.

Application Example
```c
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_https_ota_config_t ota_config = {
        .http_config = &config,
    };
    esp_err_t ret = esp_https_ota(&ota_config);
    if (ret == ESP_OK) {
        esp_restart();
    } else {
        return ESP_FAIL;
    }
    return ESP_OK;
}
```

Server Verification

Please refer to *ESP-TLS: TLS Server Verification* for more information on server verification. The root certificate in PEM format needs to be provided to the `esp_http_client_config_t::cert_pem` member.

**Note:** The server-endpoint root certificate should be used for verification instead of any intermediate ones from the certificate chain. The reason is that the root certificate has the maximum validity and usually remains the same for a long period of time. Users can also use the `esp_http_client_config_t::crt_bundle_attach` member for verification by the ESP x509 Certificate Bundle feature, which covers most of the trusted root certificates.

Partial Image Download over HTTPS

To use the partial image download feature, enable `partial_http_download` configuration in `esp_https_ota_config_t`. When this configuration is enabled, firmware image will be downloaded in multiple HTTP requests of specified sizes. Maximum content length of each request can be specified by setting `max_http_request_size` to the required value.

This option is useful while fetching image from a service like AWS S3, where mbedTLS Rx buffer size (`CONFIG_MBEDTLS_SSL_IN.Content_LEN`) can be set to a lower value which is not possible without enabling this configuration.

Default value of mbedTLS Rx buffer size is set to 16 KB. By using `partial_http_download` with `max_http_request_size` of 4 KB, size of mbedTLS Rx buffer can be reduced to 4 KB. With this configuration, memory saving of around 12 KB is expected.

Signature Verification

For additional security, signature of OTA firmware images can be verified. For more information, please refer to *Secure OTA Updates Without Secure Boot*.

Advanced APIs

`esp_https_ota` also provides advanced APIs which can be used if more information and control is needed during the OTA process.

Example that uses advanced ESP HTTPS OTA APIs: `system/ota/advanced_https_ota`.
OTA Upgrades with Pre-Encrypted Firmware

To perform OTA upgrades with pre-encrypted firmware, please enable `CONFIG_ESP_HTTPS OTA_DECRYPT_CB` in component menuconfig.

Example that performs OTA upgrade with pre-encrypted firmware: `system/ota/pre_encrypted_ota`.

OTA System Events

ESP HTTPS OTA has various events for which a handler can be triggered by the Event Loop Library when the particular event occurs. The handler has to be registered using `esp_event_handler_register()`. This helps the event handling for ESP HTTPS OTA.

`esp_https_ota_event_t` has all the events which can happen when performing OTA upgrade using ESP HTTPS OTA.

Event Handler Example

```c
/* Event handler for catching system events */
static void event_handler(void* arg, esp_event_base_t event_base, int32_t event_id, void* event_data)
{
  if (event_base == ESP_HTTPS_OTA_EVENT) {
    switch (event_id) {
      case ESP_HTTPS_OTA_START:
        ESP_LOGI(TAG, "OTA started");
        break;
      case ESP_HTTPS_OTA_CONNECTED:
        ESP_LOGI(TAG, "Connected to server");
        break;
      case ESP_HTTPS_OTA_GET_IMG_DESC:
        ESP_LOGI(TAG, "Reading Image Description");
        break;
      case ESP_HTTPS_OTA_VERIFY_CHIP_ID:
        ESP_LOGI(TAG, "Verifying chip id of new image: %d", *(esp_chip_id_t *)event_data);
        break;
      case ESP_HTTPS_OTA_DECRYPT_CB:
        ESP_LOGI(TAG, "Callback to decrypt function");
        break;
      case ESP_HTTPS_OTA_WRITE_FLASH:
        ESP_LOGD(TAG, "Writing to flash: %d written", *(int *)event_data);
        break;
      case ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION:
        ESP_LOGI(TAG, "Boot partition updated. Next Partition: %d", *(esp_partition_subtype_t *)event_data);
        break;
      case ESP_HTTPS_OTA_FINISH:
        ESP_LOGI(TAG, "OTA finish");
        break;
      case ESP_HTTPS_OTA_ABORT:
        ESP_LOGI(TAG, "OTA abort");
        break;
    }
  }
}
```

Expected data type for different ESP HTTPS OTA events in the system event loop:

- `ESP_HTTPS_OTA_START`: NULL
- `ESP_HTTPS_OTA_CONNECTED`: NULL
API Reference

Header File

- components/esp_https_ota/include/esp_https_ota.h
- This header file can be included with:

```c
#include "esp_https_ota.h"
```

- This header file is a part of the API provided by the `esp_https_ota` component. To declare that your component depends on `esp_https_ota`, add the following to your `CMakeLists.txt`:

```cmake
REQUIRES esp_https_ota
```

Functions

### `esp_err_t esp_https_ota`(const `esp_https_ota_config_t` *ota_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 `ota_config->http_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.

**Parameters**

- `ota_config` - [in] pointer to `esp_https_ota_config_t` structure.

**Returns**

- 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.

### `esp_err_t esp_https_ota_begin`(const `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.

### Parameters
- `ota_config` - [in] pointer to `esp_https_ota_config_t` structure
- `handle` - [out] pointer to an allocated data of type `esp_https_ota_handle_t` which will be initialised in this function

### Returns
- `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.

```c
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.

### Parameters
- `https_ota_handle` - [in] pointer to `esp_https_ota_handle_t` structure

### Returns
- `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_INVALID_VERSION`: Invalid chip revision in image header
- `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.

```c
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.

### Parameters
- `https_ota_handle` - [in] pointer to `esp_https_ota_handle_t` structure

### Returns
- `false`
- `true`

```c
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 esp_https_ota_finish should not be called after calling esp_https_ota_abort

### Parameters **esp_https_ota_abort**

- **esp_https_ota_handle** - [in] pointer to esp_https_ota_handle_t structure

### Returns

- **ESP_OK:** Clean-up successful
- **ESP_ERR_INVALID_STATE:** Invalid ESP HTTPS OTA state
- **ESP_FAIL:** OTA not started
- **ESP_ERR_NOT_FOUND:** OTA handle not found
- **ESP_ERR_INVALID_ARG:** Invalid argument

### Note: esp_https_ota_abort should not be called after calling esp_https_ota_finish

### Parameters **esp_https_ota_get_img_desc**

- **esp_https_ota_handle** - [in] pointer to esp_https_ota_handle_t structure
- **new_app_info** - [out] pointer to an allocated esp_app_desc_t structure

### Returns

- **ESP_ERR_INVALID_ARG:** Invalid arguments
- **ESP_ERR_INVALID_STATE:** Invalid state to call this API. esp_https_ota_begin() not called yet.
- **ESP_FAIL:** Failed to read image descriptor
- **ESP_OK:** Successfully read image descriptor

### Note: This API can be called only after esp_https_ota_begin() and before esp_https_ota_perform(). Calling this API is not mandatory.

### Parameters **esp_https_ota_get_image_len_read**

- **esp_https_ota_handle** - [in] pointer to esp_https_ota_handle_t structure

### Returns

This function returns OTA image data read so far.

### Note: This API should be called only if esp_https_ota_perform() has been called at least once or if esp_https_ota_get_img_desc has been called before.
Chapter 2. API Reference

• -1 On failure
• total bytes read so far

int esp_https_ota_get_image_size(esp_https_ota_handle_t https_ota_handle)
This function returns OTA image total size.

Note: This API should be called after esp_https_ota_begin() has been already called. This can be used to create some sort of progress indication (in combination with esp_https_ota_get_image_len_read())

Parameters

https_ota_handle - [in] pointer to esp_https_ota_handle_t structure

Returns

• -1 On failure or chunked encoding
• total bytes of image

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

http_client_init_cb_t http_client_init_cb

Callback after ESP HTTP client is initialised

bool bulk_flash_erase

Erase entire flash partition during initialization. By default flash partition is erased during write operation and in chunk of 4K sector size

bool partial_http_download

Enable Firmware image to be downloaded over multiple HTTP requests

int max_http_request_size

Maximum request size for partial HTTP download

Macros

ESP_ERR_HTTPS_OTA_BASE

ESP_ERR_HTTPS_OTA_IN_PROGRESS

Type Definitions

typedef void *esp_https_ota_handle_t

typedef esp_err_t (*http_client_init_cb_t)(esp_http_client_handle_t)
Chapter 2. API Reference

Enumerations

def \texttt{esp\_https\_ota\_event\_t}

Events generated by OTA process.

\textit{Values:}

- \texttt{ESP\_HTTPS\_OTA\_START}
  
  OTA started

- \texttt{ESP\_HTTPS\_OTA\_CONNECTED}
  
  Connected to server

- \texttt{ESP\_HTTPS\_OTA\_GET\_IMG\_DESC}
  
  Read app description from image header

- \texttt{ESP\_HTTPS\_OTA\_VERIFY\_CHIP\_ID}
  
  Verify chip id of new image

- \texttt{ESP\_HTTPS\_OTA\_DECRYPT\_CB}
  
  Callback to decrypt function

- \texttt{ESP\_HTTPS\_OTA\_WRITE\_FLASH}
  
  Flash write operation

- \texttt{ESP\_HTTPS\_OTA\_UPDATE\_BOOT\_PARTITION}
  
  Boot partition update after successful ota update

- \texttt{ESP\_HTTPS\_OTA\_FINISH}
  
  OTA finished

- \texttt{ESP\_HTTPS\_OTA\_ABORT}
  
  OTA aborted

2.9.10 Event Loop Library

Overview

The event loop library allows components to declare events so that other components can register handlers –codes that executes when those events occur. This allows loosely-coupled components to attach desired behavior to state changes of other components without application involvement. This also simplifies event processing by serializing and deferring code execution to another context.

One common case is, if a high-level library is using the Wi-Fi library: it may subscribe to \textit{ESP32 Wi-Fi Programming Model} directly and act on those events.
Chapter 2. API Reference

Using esp_event APIs

There are two objects of concern for users of this library: events and event loops.

An event indicates an important occurrence, such as a successful Wi-Fi connection to an access point. A two-part identifier should be used when referencing events, see declaring and defining events for details. The event loop is the bridge between events and event handlers. The event source publishes events to the event loop using the APIs provided by the event loop library, and event handlers registered to the event loop respond to specific types of events.

Using this library roughly entails the following flow:

1. The user defines a function that should run when an event is posted to a loop. This function is referred to as the event handler, and 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 is discussed in default event loop.
3. Components register event handlers to the loop using esp_event_handler_register_with(). Handlers can be registered with multiple loops, see notes on handler registration.
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 that are no longer needed can be deleted using esp_event_loop_delete().

In code, the flow above may look like as follows:

```c
// 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.
    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 specify a hypothetical event that handler run_on_event should execute 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, the event loop executes the event handler registered to the posted event, in this case, run_on_event. To simplify the process, this example calls esp_event_post_to from app_main, but posting can be done from any other task (which is the more interesting use case).
    ...
```
Declaring and Defining Events

As mentioned previously, events consist 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:

```c
ESP_EVENT_DECLARE_BASE(EVENT_BASE)
```

Event base definition:

```c
ESP_EVENT_DEFINE_BASE(EVENT_BASE)
```

Note: In ESP-IDF, the base identifiers for system events are uppercase and are postfixed with _EVENT. For example, the base for Wi-Fi 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 IDs, declaring them as enumerations is recommended. Once again, for visibility, these are typically placed in public header files.

Event ID:

```c
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 (Wi-Fi events, for example). The handle for this loop is hidden from the user, and the creation, deletion, handler registration/deregistration, and posting of events are done through a variant of the APIs for user event loops. The table below enumerates those variants, and the user event loops equivalent.
Chapter 2. API Reference

### User Event Loops

<table>
<thead>
<tr>
<th>User Event Loops</th>
<th>Default Event Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp_event_loop_create()</td>
<td>esp_event_loop_create_default()</td>
</tr>
<tr>
<td>esp_event_loop_delete()</td>
<td>esp_event_loop_delete_default()</td>
</tr>
<tr>
<td>esp_event_handler_register_with()</td>
<td>esp_event_handler_register()</td>
</tr>
<tr>
<td>esp_event_handler_unregister_with()</td>
<td>esp_event_handler_unregister()</td>
</tr>
<tr>
<td>esp_event_post_to()</td>
<td>esp_event_post()</td>
</tr>
</tbody>
</table>

If you compare the signatures for both, they are mostly similar except 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, there is no difference in 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 by 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 the following situations:

1. all events that get posted to a loop
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 Un-Registering Itself**  In general, an event handler run by an event loop is not allowed to do any registering/unregistering activity on that event loop. There is one exception, though: un-registering itself is allowed for the handler. E.g., it is possible to do the following:
**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 get 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 also gets its handlers registered first. Handlers registered one after the other by a single task are still dispatched in the order relative to each other, but if that task gets pre-empted in between registration by another task that also registers handlers; then during dispatch those handlers 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 of using the `esp_event` library can be found in `system/esp_event`. The examples cover event declaration, loop creation, handler registration and deregistration, and event posting.

Other examples which also adopt `esp_event` library:

- NMEA Parser, which decodes the statements received from GPS.

**API Reference**

**Header File**

- `components/esp_event/include/esp_event.h`

  This header file can be included with:

  ```c
  #include "esp_event.h"
  ```

  This header file is a part of the API provided by the `esp_event` component. To declare that your component depends on `esp_event`, add the following to your `CMakeLists.txt`:

  ```cmake
  Requires esp_event
  ```

  or

```c
void run_on_event (void* handler_arg, esp_event_base_t base, int32_t id, void* __event_data) {
    esp_event_loop_handle_t *loop_handle = (esp_event_loop_handle_t*) handler_arg;
    esp_event_handler_unregister_with (loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_on_event);
}

void app_main (void) {
    esp_event_loop_handle_t loop_handle;
    esp_event_loop_create (&loop_args, &loop_handle);
    esp_event_handler_register_with (loop_handle, MY_EVENT_BASE, MY_EVENT_ID, run_on_event, &loop_handle);
    // ... post-event MY_EVENT_BASE, MY_EVENT_ID and run loop at some point
```
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.

**Parameters**

- **event_loop_args**  ⚱  [in] configuration structure for the event loop to create
- **event_loop**  ⚱  [out] handle to the created event loop

**Returns**

- 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

**esp_err_t esp_event_loop_delete**(esp_event_loop_handle_t event_loop)

Delete an existing event loop.

**Parameters**

- **event_loop**  ⚱  [in] event loop to delete, must not be NULL

**Returns**

- ESP_OK: Success
- Others: Fail

**esp_err_t esp_event_loop_create_default**(void)

Create default event loop.

**Returns**

- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for event loops list
- ESP_ERR_INVALID_STATE: Default event loop has already been created
- ESP_FAIL: Failed to create task loop
- Others: Fail

**esp_err_t esp_event_loop_delete_default**(void)

Delete the default event loop.

**Returns**

- 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 the allotted time waiting on the event queue. Once an event has been dequeued, 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 dequeued event, and (2) during dispatch of the dequeued 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 dequeued event.

In cases where waiting on the queue times out, ESP_OK is returned and not ESP_ERR_TIMEOUT, since it is normal behavior.
Chapter 2. API Reference

**Note:** encountering an unknown event that has been posted to the loop will only generate a warning, not an error.

**Parameters**
- `event_loop` - [in] event loop to dispatch posted events from, must not be NULL
- `ticks_to_run` - [in] number of ticks to run the loop

**Returns**
- ESP_OK: Success
- Others: Fail

```c
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 (legacy).

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

**Parameters**
- `event_base` - [in] the base ID of the event to register the handler for
- `event_id` - [in] the ID of the event to register the handler for
- `event_handler` - [in] the handler function which gets called when the event is dispatched
- `event_handler_arg` - [in] data, aside from event data, that is passed to the handler when it is called

**Returns**
- 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

```c
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 (legacy).

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
Chapter 2. API Reference

Parameters

- `event_loop` [in] the event loop to register this handler function to, must not be NULL
- `event_base` [in] the base ID of the event to register the handler for
- `event_id` [in] the ID of the event to register the handler for
- `event_handler` [in] the handler function which gets called when the event is dispatched
- `event_handler_arg` [in] data, aside from event data, that is passed to the handler when it is called

Returns

- 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

```c
esp_err_t esp_event_handler_instance_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, 
    esp_event_handler_instance_t* instance)
```

Register an instance of event handler to a specific 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

Besides the error, the function returns an instance object as output parameter to identify each registration. This is necessary to remove (unregister) the registration before the event loop is deleted.

Registering multiple handlers to events, registering a single handler to multiple events as well as registering the same handler to the same event multiple times is possible. Each registration yields a distinct instance object which identifies it over the registration lifetime.

**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

Parameters

- `event_loop` [in] the event loop to register this handler function to, must not be NULL
- `event_base` [in] the base ID of the event to register the handler for
- `event_id` [in] the ID of the event to register the handler for
- `event_handler` [in] the handler function which gets called when the event is dispatched
- `event_handler_arg` [in] data, aside from event data, that is passed to the handler when it is called
- `instance` [out] An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same for all registrations. If no unregistration is needed, but the handler should be deleted when the event loop is deleted, instance can be NULL.

Returns

- 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 or instance is NULL
• Others: Fail

```c
esp_err_t esp_event_handler_instance_register(esp_event_base_t event_base, int32_t event_id, esp_event_handler_t event_handler, void *event_handler_arg, esp_event_handler_instance_t *instance)
```

Register an instance of event handler to the default loop.
This function does the same as esp_event_handler_instance_register_with, except that it registers the handler to the default event loop.

**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

### Parameters

- **event_base** - [in] the base ID of the event to register the handler for
- **event_id** - [in] the ID of the event to register the handler for
- **event_handler** - [in] the handler function which gets called when the event is dispatched
- **event_handler_arg** - [in] data, aside from event data, that is passed to the handler when it is called
- **instance** - [out] An event handler instance object related to the registered event handler and data, can be NULL. This needs to be kept if the specific callback instance should be unregistered before deleting the whole event loop. Registering the same event handler multiple times is possible and yields distinct instance objects. The data can be the same for all registrations. If no unregistration is needed, but the handler should be deleted when the event loop is deleted, instance can be NULL.

### Returns

- 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 or instance is NULL
- Others: Fail

```c
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 (legacy).
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.

### Parameters

- **event_base** - [in] the base of the event with which to unregister the handler
- **event_id** - [in] the ID of the event with which to unregister the handler
- **event_handler** - [in] the handler to unregister

### Returns

ESP_OK success
Returns ESP_ERR_INVALID_ARG invalid combination of event base and event ID
Returns others fail

```
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 from a specific event loop (legacy).
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.

Parameters
- `event_loop` - [in] the event loop with which to unregister this handler function, must not be NULL
- `event_base` - [in] the base of the event with which to unregister the handler
- `event_id` - [in] the ID of the event with which to unregister the handler
- `event_handler` - [in] the handler to unregister

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

```
esp_err_t esp_event_handler_instance_unregister_with(esp_event_loop_handle_t event_loop,
                                                   esp_event_base_t event_base, int32_t event_id,
                                                   esp_event_handler_instance_t instance)
```

Unregister a handler instance from a specific event loop.

Unregisters a handler instance, so it will no longer be called during dispatch. Handler instances can be unregistered for any combination of event_base and event_id which were previously registered. To unregister a handler instance, the event_base and event_id arguments must match exactly the arguments passed to esp_event_handler_instance_register() when that handler instance was registered. Passing ESP_EVENT_ANY_BASE and/or ESP_EVENT_ANY_ID will only unregister handler instances 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.

Parameters
- `event_loop` - [in] the event loop with which to unregister this handler function, must not be NULL
- `event_base` - [in] the base of the event with which to unregister the handler
- `event_id` - [in] the ID of the event with which to unregister the handler
- `instance` - [in] the instance object of the registration to be unregistered

Returns
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

```
esp_err_t esp_event_handler_instance_unregister(es_event_base_t event_base, int32_t event_id,
                                             esp_event_handler_instance_t instance)
```

Unregister a handler from the system event loop.
This function does the same as esp_event_handler_instance_unregister_with, except that it unregisters the handler instance from the default event loop.

Parameters
- `event_base` - [in] the base of the event with which to unregister the handler
Chapter 2. API Reference

- **event_id** - [in] the ID of the event with which to unregister the handler
- **instance** - [in] the instance object of the registration to be unregistered

**Returns**
- ESP_OK: Success
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID
- Others: Fail

```c
esp_err_t esp_event_post (esp_event_base_t event_base, int32_t event_id, const 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.

**Parameters**
- **event_base** - [in] the event base that identifies the event
- **event_id** - [in] the event ID that identifies the event
- **event_data** - [in] the data, specific to the event occurrence, that gets passed to the handler
- **event_data_size** - [in] the size of the event data
- **ticks_to_wait** - [in] number of ticks to block on a full event queue

**Returns**
- 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

```c
esp_err_t esp_event_post_to (esp_event_loop_handle_t event_loop, esp_event_base_t event_base, int32_t event_id, const 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.

**Parameters**
- **event_loop** - [in] the event loop to post to, must not be NULL
- **event_base** - [in] the event base that identifies the event
- **event_id** - [in] the event ID that identifies the event
- **event_data** - [in] the data, specific to the event occurrence, that gets passed to the handler
- **event_data_size** - [in] the size of the event data
- **ticks_to_wait** - [in] number of ticks to block on a full event queue

**Returns**
- 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

```c
esp_err_t esp_event_isr_post (esp_event_base_t event_base, int32_t event_id, const void *event_data, size_t event_data_size, BaseType_t *task_unblocked)
```

Special variant of `esp_event_post` for posting events from interrupt handlers.

**Note:** this function is only available when CONFIG_ESP_EVENT_POST_FROM_ISR is enabled
Note: when this function is called from an interrupt handler placed in IRAM, this function should be placed in IRAM as well by enabling CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR

Parameters
- `event_base` - [in] the event base that identifies the event
- `event_id` - [in] the event ID that identifies the event
- `event_data` - [in] the data, specific to the event occurrence, that gets passed to the handler
- `event_data_size` - [in] the size of the event data; max is 4 bytes
- `task_unblocked` - [out] an optional parameter (can be NULL) which indicates that an event task with higher priority than currently running task has been unblocked by the posted event; a context switch should be requested before the interrupt is existed.

Returns
- ESP_OK: Success
- ESP_FAIL: Event queue for the default event loop full
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID, data size of more than 4 bytes
- Others: Fail

`esp_err_t esp_event_isr_post_to(esp_event_loop_handle_t event_loop, esp_event_base_t event_base, int32_t event_id, const void* event_data, size_t event_data_size, BaseType_t* task_unblocked)`

Special variant of esp_event_post_to for posting events from interrupt handlers.

Note: this function is only available when CONFIG_ESP_EVENT_POST_FROM_ISR is enabled

Note: when this function is called from an interrupt handler placed in IRAM, this function should be placed in IRAM as well by enabling CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR

Parameters
- `event_loop` - [in] the event loop to post to, must not be NULL
- `event_base` - [in] the event base that identifies the event
- `event_id` - [in] the event ID that identifies the event
- `event_data` - [in] the data, specific to the event occurrence, that gets passed to the handler
- `event_data_size` - [in] the size of the event data
- `task_unblocked` - [out] an optional parameter (can be NULL) which indicates that an event task with higher priority than currently running task has been unblocked by the posted event; a context switch should be requested before the interrupt is existed.

Returns
- ESP_OK: Success
- ESP_FAIL: Event queue for the loop full
- ESP_ERR_INVALID_ARG: Invalid combination of event base and event ID, data size of more than 4 bytes
- Others: Fail

`esp_err_t esp_event_dump(FILE* file)`

Dumps statistics of all event loops.

Dumps event loop info in the format:
event loop
counter
... 
where:

event loopormat: address, name rx:total_received dr:total_dropped
where:
address - memory address of the event loop
name - name of the event loop, 'none' if no dedicated task
total_received - number of successfully posted events
total_dropped - number of events unsuccessfully posted due to queue...becoming full

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

Parameters file - [in] the file stream to output to

Returns
- ESP_OK: Success
- ESP_ERR_NO_MEM: Cannot allocate memory for event loops list
- Others: Fail

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 \textbf{task_stack_size} \\
stack size of the event loop task, ignored if task name is NULL

BaseType_t \textbf{task_core_id} \\
core to which the event loop task is pinned to, ignored if task name is NULL

\section*{Header File}

- \texttt{components/esp_event/include/esp_event_base.h}
- This header file can be included with:

\begin{verbatim}
#include "esp_event_base.h"
\end{verbatim}

- This header file is a part of the API provided by the \texttt{esp_event} component. To declare that your component depends on \texttt{esp_event}, add the following to your CMakeLists.txt:

\begin{verbatim}
requires esp_event
\end{verbatim}

or

\begin{verbatim}
priv_requires esp_event
\end{verbatim}

\section*{Macros}

\texttt{ESP\_EVENT\_DECLARE\_BASE} (id)  
\texttt{ESP\_EVENT\_DEFINE\_BASE} (id)  
\texttt{ESP\_EVENT\_ANY\_BASE}  
\hspace{1em} register handler for any event base

\texttt{ESP\_EVENT\_ANY\_ID}  
\hspace{1em} register handler for any event id

\section*{Type Definitions}

typedef void *\texttt{esp_event_loop_handle_t}  
a number that identifies an event with respect to a base

typedef void (*\texttt{esp_event_handler_t}) (void *event_handler_arg, esp_event_base_t event_base, int32_t event_id, void *event_data)  
\hspace{1em} function called when an event is posted to the queue

typedef void *\texttt{esp_event_handler_instance_t}  
\hspace{1em} context identifying an instance of a registered event handler

\section*{Related Documents}

2.9.11 FreeRTOS Overview
Chapter 2. API Reference

Overview

FreeRTOS is an open source RTOS (real-time operating system) kernel that is integrated into ESP-IDF as a component. Thus, all ESP-IDF applications and many ESP-IDF components are written based on FreeRTOS. The FreeRTOS kernel is ported to all architectures (i.e., Xtensa and RISC-V) available of ESP chips.

Furthermore, ESP-IDF provides different implementations of FreeRTOS in order to support SMP (Symmetric Multiprocessing) on multi-core ESP chips. This document provides an overview of the FreeRTOS component, the different FreeRTOS implementations offered by ESP-IDF, and the common aspects across all implementations.

Implementations

The official FreeRTOS (henceforth referred to as Vanilla FreeRTOS) is a single-core RTOS. In order to support the various multi-core ESP targets, ESP-IDF supports different FreeRTOS implementations as listed below:

**ESP-IDF FreeRTOS** ESP-IDF FreeRTOS is a FreeRTOS implementation based on Vanilla FreeRTOS v10.5.1, but contains significant modifications to support SMP. ESP-IDF FreeRTOS only supports two cores at most (i.e., dual core SMP), but is more optimized for this scenario by design. For more details regarding ESP-IDF FreeRTOS and its modifications, please refer to the FreeRTOS (ESP-IDF) document.

**Amazon SMP FreeRTOS** Amazon SMP FreeRTOS is an SMP implementation of FreeRTOS that is officially supported by Amazon. Amazon SMP FreeRTOS is able to support N-cores (i.e., more than two cores). Amazon SMP FreeRTOS can be enabled via the CONFIG_FREERTOS_SMP option. For more details regarding Amazon SMP FreeRTOS, please refer to the official Amazon SMP FreeRTOS documentation.

**Warning:** The Amazon SMP FreeRTOS implementation (and its port in ESP-IDF) are currently in experimental/beta state. Therefore, significant behavioral changes and breaking API changes can occur.

Configuration

**Kernel Configuration** Vanilla FreeRTOS requires that ports and applications configure the kernel by adding various #define config... macro definitions to the FreeRTOSConfig.h header file. Vanilla FreeRTOS supports a list of kernel configuration options which allow various kernel behaviors and features to be enabled or disabled.

However, for all FreeRTOS ports in ESP-IDF, the FreeRTOSConfig.h header file is considered private and must not be modified by users. A large number of kernel configuration options in FreeRTOSConfig.h are hard-coded as they are either required/not supported by ESP-IDF. All kernel configuration options that are configurable by the user are exposed via menuconfig under Component Config/FreeRTOS/Kernel.

For the full list of user configurable kernel options, see Project Configuration. The list below highlights some commonly used kernel configuration options:

- **CONFIG_FREERTOS_UNICORE** runs FreeRTOS only on CPU0. Note that this is not equivalent to running Vanilla FreeRTOS. Furthermore, this option may affect behavior of components other than freertos. For more details regarding the effects of running FreeRTOS on a single core, refer to ESP-IDF FreeRTOS Single Core (if using ESP-IDF FreeRTOS) or the official Amazon SMP FreeRTOS documentation. Alternatively, users can also search for occurrences of CONFIG_FREERTOS_UNICORE in the ESP-IDF components.

**Note:** As ESP32-S2 is a single core SoC, the CONFIG_FREERTOS_UNICORE configuration is always set.
• **CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY** enables backward compatibility with some FreeRTOS macros/types/functions that were deprecated from v8.0 onwards.

**Port Configuration** All other FreeRTOS related configuration options that are not part of the kernel configuration are exposed via menuconfig under Component Config/FreeRTOS/Port. These options configure aspects such as:

- The FreeRTOS ports themselves (e.g., tick timer selection, ISR stack size)
- Additional features added to the FreeRTOS implementation or ports

**Using FreeRTOS**

**Application Entry Point** Unlike Vanilla FreeRTOS, users of FreeRTOS in ESP-IDF must never call `vTaskStartScheduler()` and `vTaskEndScheduler()`. Instead, ESP-IDF starts FreeRTOS automatically. Users must define a `void app_main(void)` function which acts as the entry point for user’s application and is automatically invoked on ESP-IDF startup.

- Typically, users would spawn the rest of their application’s task from `app_main`.
- The `app_main` function is allowed to return at any point (i.e., before the application terminates).
- The `app_main` function is called from the main task.

**Background Tasks** During startup, ESP-IDF and the FreeRTOS kernel automatically create multiple tasks that run in the background (listed in the table below).

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Description</th>
<th>Stack Min Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle Tasks (IDLEx)</td>
<td>An idle task (IDLEx) is created for (and pinned to) each CPU core, where x is the CPU core’s number. The x is dropped when single-core configuration is enabled.</td>
<td>CONFIG_FREERTOS_IDLE_TASK_STACKSIZE Core x 0</td>
</tr>
<tr>
<td>FreeRTOS Timer Task (Tmr Svc)</td>
<td>FreeRTOS will create the Timer Service/Daemon Task if any FreeRTOS Timer APIs are called by the application</td>
<td>CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH Core 0</td>
</tr>
<tr>
<td>Main Task (main)</td>
<td>Task that simply calls <code>app_main</code>. This task will self delete when <code>app_main</code> returns</td>
<td>CONFIG_ESP_MAIN_TASK_STACK_SIZE CONFIG_ESP_MAIN_TASK_AFFINITY 1</td>
</tr>
<tr>
<td>IPC Tasks (ipcx)</td>
<td>When CONFIG_FREERTOS_UNICORE is false, an IPC task (ipcx) is created for (and pinned to) each CPU core. IPC tasks are used to implement the Inter-processor Call (IPC) feature.</td>
<td>CONFIG_FREERTOS_IPC_TASK_STACK_SIZE Core 24</td>
</tr>
<tr>
<td>ESP Timer Task (esp_timer)</td>
<td>ESP-IDF creates the ESP Timer Task used to process ESP Timer callbacks</td>
<td>CONFIG_FREERTOS_TIMER_TASK_STACK_SIZE Core 22</td>
</tr>
</tbody>
</table>

**Note:** Note that if an application uses other ESP-IDF features (e.g., Wi-Fi or Bluetooth), those features may create their own background tasks in addition to the tasks listed in the table above.
FreeRTOS Additions

ESP-IDF provides some supplemental features to FreeRTOS such as Ring Buffers, ESP-IDF style Tick and Idle Hooks, and TLSP deletion callbacks. See FreeRTOS (Supplemental Features) for more details.

FreeRTOS Heap

Vanilla FreeRTOS provides its own selection of heap implementations. However, ESP-IDF already implements its own heap (see Heap Memory Allocation), thus ESP-IDF does not make use of the heap implementations provided by Vanilla FreeRTOS. All FreeRTOS ports in ESP-IDF map FreeRTOS memory allocation or free calls (e.g., pvPortMalloc() and pvPortFree()) to ESP-IDF heap API (i.e., heap_caps_malloc() and heap_caps_free()). However, the FreeRTOS ports ensure that all dynamic memory allocated by FreeRTOS is placed in internal memory.

Note: If users wish to place FreeRTOS tasks/objects in external memory, users can use the following methods:

- Allocate the task or object using one of the ...CreateWithCaps() API, such as xTaskCreateWithCaps() and xQueueCreateWithCaps() (see IDF Additional API for more details).
- Manually allocate external memory for those objects using heap_caps_malloc(), then create the objects from the allocated memory using on of the ...CreateStatic() FreeRTOS functions.

2.9.12 FreeRTOS (ESP-IDF)

Overview

The original FreeRTOS (hereinafter referred to as Vanilla FreeRTOS) is a compact and efficient real-time operating system supported on many single-core MCUs and SoCs. However, to support numerous dual-core ESP targets, such as ESP32, ESP32-S3, and ESP32-P4, ESP-IDF provides an implementation of FreeRTOS with dual-core symmetric multiprocessing (SMP) capabilities (hereinafter referred to as ESP-IDF FreeRTOS).

ESP-IDF FreeRTOS is based on Vanilla FreeRTOS v10.5.1 but contains significant modifications to both API and kernel behavior in order to support dual-core SMP. This document describes the API and behavioral differences between Vanilla FreeRTOS and ESP-IDF FreeRTOS.

Note: This document assumes that the reader has a requisite understanding of Vanilla FreeRTOS, i.e., its features, behavior, and API usage. Refer to the Vanilla FreeRTOS documentation for more details.

Note: ESP-IDF FreeRTOS can be built for a single core by enabling the CONFIG_FREERTOS_UNICORE configuration option. ESP targets that are single core always have the CONFIG_FREERTOS_UNICORE option enabled. However, note that building with CONFIG_FREERTOS_UNICORE enabled does not equate to building with Vanilla FreeRTOS, as some of the behavioral and API changes of ESP-IDF are still present. For more details, see ESP-IDF FreeRTOS Single Core.

This document is split into the following parts.

Contents

- FreeRTOS (ESP-IDF)
  - Overview
Symmetric Multiprocessing

Basic Concepts  Symmetric multiprocessing is a computing architecture where two or more identical CPUs (cores) are connected to a single shared main memory and controlled by a single operating system. In general, an SMP system:

• has multiple cores running independently. Each core has its own register file, interrupts, and interrupt handling.
• presents an identical view of memory to each core. Thus, a piece of code that accesses a particular memory address has the same effect regardless of which core it runs on.

The main advantages of an SMP system compared to single-core or asymmetric multiprocessing systems are that:

• the presence of multiple CPUs allows for multiple hardware threads, thus increasing overall processing throughput.
• having symmetric memory means that threads can switch cores during execution. This, in general, can lead to better CPU utilization.

Although an SMP system allows threads to switch cores, there are scenarios where a thread must or should only run on a particular core. Therefore, threads in an SMP system also have a core affinity that specifies which particular core the thread is allowed to run on.

• A thread that is pinned to a particular core is only able to run on that core.
• A thread that is unpinned will be allowed to switch between cores during execution instead of being pinned to a particular core.

SMP on an ESP Target  ESP targets such as ESP32, ESP32-S3, and ESP32-P4 are dual-core SMP SoCs. These targets have the following hardware features that make them SMP-capable:

• Two identical cores are known as CPU0 and CPU1. This means that the execution of a piece of code is identical regardless of which core it runs on.
• Symmetric memory (with some small exceptions).
  – If multiple cores access the same memory address simultaneously, their access will be serialized by the memory bus.
  – True atomic access to the same memory address is achieved via an atomic compare-and-swap instruction provided by the ISA.
• Cross-core interrupts that allow one CPU to trigger an interrupt on another CPU. This allows cores to signal each other.

Note:  CPU0 is also known as Protocol CPU or PRO_CPU and CPU1 is also known as Application CPU or APP_CPU. The PRO_CPU and APP_CPU aliases for CPU0 and CPU1 exist in ESP-IDF as they reflect how typical ESP-IDF applications utilize the two CPUs. Typically, the tasks responsible for handling wireless networking (e.g., Wi-Fi or Bluetooth) are pinned to CPU0, thus the name PRO_CPU; whereas the tasks handling the remainder of the application are pinned to CPU1, thus the name APP_CPU.

Tasks

Creation  Vanilla FreeRTOS provides the following functions to create a task:

• xTaskCreate() creates a task. The task’s memory is dynamically allocated.
• xTaskCreateStatic() creates a task. The task’s memory is statically allocated, i.e., provided by the user.
However, in an SMP system, tasks need to be assigned a particular affinity. Therefore, ESP-IDF provides a **Pinned-ToCore** version of Vanilla FreeRTOS’s task creation functions:

- `xTaskCreatePinnedToCore()` creates a task with a particular core affinity. The task’s memory is dynamically allocated.
- `xTaskCreateStaticPinnedToCore()` creates a task with a particular core affinity. The task’s memory is statically allocated, i.e., provided by the user.

The PinnedToCore versions of the task creation function API differ from their vanilla counterparts by having an extra `xCOREID` parameter that is used to specify the created task’s core affinity. The valid values for core affinity are:

- 0, which pins the created task to CPU0
- 1, which pins the created task to CPU1
- `tskNO_AFFINITY`, which allows the task to be run on both CPUs

Note that ESP-IDF FreeRTOS still supports the vanilla versions of the task creation functions. However, these standard functions have been modified to essentially invoke their respective PinnedToCore counterparts while setting the core affinity to `tskNO_AFFINITY`.

**Note:** ESP-IDF FreeRTOS also changes the units of `ulStackDepth` in the task creation functions. Task stack sizes in Vanilla FreeRTOS are specified in a number of words, whereas in ESP-IDF FreeRTOS, the task stack sizes are specified in bytes.

---

**Execution**  
The anatomy of a task in ESP-IDF FreeRTOS is the same as in Vanilla FreeRTOS. More specifically, ESP-IDF FreeRTOS tasks:

- Can only be in one of the following states: Running, Ready, Blocked, or Suspended.
- Task functions are typically implemented as an infinite loop.
- Task functions should never return.

**Deletion**  
Task deletion in Vanilla FreeRTOS is called via `vTaskDelete()`. The function allows deletion of another task or the currently running task if the provided task handle is NULL. The actual freeing of the task’s memory is sometimes delegated to the idle task if the task being deleted is the currently running task.

ESP-IDF FreeRTOS provides the same `vTaskDelete()` function. However, due to the dual-core nature, there are some behavioral differences when calling `vTaskDelete()` in ESP-IDF FreeRTOS:

- When deleting a task that is currently running on the other core, a yield is triggered on the other core, and the task’s memory is freed by one of the idle tasks.
- A deleted task’s memory is freed immediately if it is not running on either core.

Please avoid deleting a task that is running on another core as it is difficult to determine what the task is performing, which may lead to unpredictable behavior such as:

- Deleting a task that is holding a mutex.
- Deleting a task that has yet to free memory it previously allocated.

Where possible, please design your own application so that when calling `vTaskDelete()`, the deleted task is in a known state. For example:

- Tasks self-deleting via `vTaskDelete(NULL)` when their execution is complete and have also cleaned up all resources used within the task.
- Tasks placing themselves in the suspend state via `vTaskSuspend()` before being deleted by another task.

**SMP Scheduler**

The Vanilla FreeRTOS scheduler is best described as a **fixed priority preemptive scheduler with time slicing** meaning that:
• Each task is given a constant priority upon creation. The scheduler executes the highest priority ready-state task.
• The scheduler can switch execution to another task without the cooperation of the currently running task.
• The scheduler periodically switches execution between ready-state tasks of the same priority in a round-robin fashion. Time slicing is governed by a tick interrupt.

The ESP-IDF FreeRTOS scheduler supports the same scheduling features, i.e., Fixed Priority, Preemption, and Time Slicing, albeit with some small behavioral differences.

**Fixed Priority** In Vanilla FreeRTOS, when the scheduler selects a new task to run, it always selects the current highest priority ready-state task. In ESP-IDF FreeRTOS, each core independently schedules tasks to run. When a particular core selects a task, the core will select the highest priority ready-state task that can be run by the core. A task can be run by the core if:

• The task has a compatible affinity, i.e., is either pinned to that core or is unpinned.
• The task is not currently being run by another core.

However, please do not assume that the two highest priority ready-state tasks are always run by the scheduler, as a task’s core affinity must also be accounted for. For example, given the following tasks:

• Task A of priority 10 pinned to CPU0
• Task B of priority 9 pinned to CPU0
• Task C of priority 8 pinned to CPU1

The resulting schedule will have Task A running on CPU0 and Task C running on CPU1. Task B is not run even though it is the second-highest priority task.

**Preemption** In Vanilla FreeRTOS, the scheduler can preempt the currently running task if a higher priority task becomes ready to execute. Likewise in ESP-IDF FreeRTOS, each core can be individually preempted by the scheduler if the scheduler determines that a higher-priority task can run on that core.

However, there are some instances where a higher-priority task that becomes ready can be run on multiple cores. In this case, the scheduler only preempts one core. The scheduler always gives preference to the current core when multiple cores can be preempted. In other words, if the higher priority ready task is unpinned and has a higher priority than the current priority of both cores, the scheduler will always choose to preempt the current core. For example, given the following tasks:

• Task A of priority 8 currently running on CPU0
• Task B of priority 9 currently running on CPU1
• Task C of priority 10 that is unpinned and was unblocked by Task B

The resulting schedule will have Task A running on CPU0 and Task C preempting Task B given that the scheduler always gives preference to the current core.

**Time Slicing** The Vanilla FreeRTOS scheduler implements time slicing, which means that if the current highest priority contains multiple ready tasks, the scheduler will switch between those tasks periodically in a round-robin fashion.

However, in ESP-IDF FreeRTOS, it is not possible to implement perfect Round Robin time slicing due to the fact that a particular task may not be able to run on a particular core due to the following reasons:

• The task is pinned to another core.
• For unpinned tasks, the task is already being run by another core.

Therefore, when a core searches the ready-state task list for a task to run, the core may need to skip over a few tasks in the same priority list or drop to a lower priority in order to find a ready-state task that the core can run.

The ESP-IDF FreeRTOS scheduler implements a Best Effort Round Robin time slicing for ready-state tasks of the same priority by ensuring that tasks that have been selected to run are placed at the back of the list, thus giving unselected tasks a higher priority on the next scheduling iteration (i.e., the next tick interrupt or yield).

The following example demonstrates the Best Effort Round Robin time slicing in action. Assume that:
• There are four ready-state tasks of the same priority AX, B0, C1, and D1 where:
  – The priority is the current highest priority with ready-state.
  – The first character represents the task’s name, i.e., A, B, C, D.
  – The second character represents the task’s core pinning, and X means unpinned.
• The task list is always searched from the head.

1. Starting state. None of the ready-state tasks have been selected to run.

   Head [ AX, B0, C1, D0 ] Tail

2. Core 0 has a tick interrupt and searches for a task to run.
   
   Task A is selected and moved to the back of the list.

   Core0--|
   Head [ AX, B0, C1, D0 ] Tail

   Head [ B0, C1, D0, AX ] Tail

3. Core 1 has a tick interrupt and searches for a task to run.
   
   Task B cannot be run due to incompatible affinity, so core 1 skips to Task C.
   
   Task C is selected and moved to the back of the list.

   Core1------|
   Head [ B0, C1, D0, AX ] Tail

   Head [ B0, D0, AX, C1 ] Tail

4. Core 0 has another tick interrupt and searches for a task to run.
   
   Task B is selected and moved to the back of the list.

   Core0--|
   Head [ B0, D0, AX, C1 ] Tail

   Head [ D0, AX, C1, B0 ] Tail

5. Core 1 has another tick and searches for a task to run.
   
   Task D cannot be run due to incompatible affinity, so core 1 skips to Task A.
   
   Task A is selected and moved to the back of the list.

   Core1------|
   Head [ D0, AX, C1, B0 ] Tail

(continues on next page)
Chapter 2. API Reference

The implications to users regarding the Best Effort Round Robin time slicing:

• Users cannot expect multiple ready-state tasks of the same priority to run sequentially as is the case in Vanilla FreeRTOS. As demonstrated in the example above, a core may need to skip over tasks.
• However, given enough ticks, a task will eventually be given some processing time.
• If a core cannot find a task runnable task at the highest ready-state priority, it will drop to a lower priority to search for tasks.
• To achieve ideal round-robin time slicing, users should ensure that all tasks of a particular priority are pinned to the same core.

Tick Interrupts  Vanilla FreeRTOS requires that a periodic tick interrupt occurs. The tick interrupt is responsible for:

• Incrementing the scheduler’s tick count
• Unblocking any blocked tasks that have timed out
• Checking if time slicing is required, i.e., triggering a context switch
• Executing the application tick hook

In ESP-IDF FreeRTOS, each core receives a periodic interrupt and independently runs the tick interrupt. The tick interrupts on each core are of the same period but can be out of phase. However, the tick responsibilities listed above are not run by all cores:

• CPU0 executes all of the tick interrupt responsibilities listed above
• CPU1 only checks for time slicing and executes the application tick hook

Note: CPU0 is solely responsible for keeping time in ESP-IDF FreeRTOS. Therefore, anything that prevents CPU0 from incrementing the tick count, such as suspending the scheduler on CPU0, will cause the entire scheduler’s timekeeping to lag behind.

Idle Tasks  Vanilla FreeRTOS will implicitly create an idle task of priority 0 when the scheduler is started. The idle task runs when no other task is ready to run, and it has the following responsibilities:

• Freeing the memory of deleted tasks
• Executing the application idle hook

In ESP-IDF FreeRTOS, a separate pinned idle task is created for each core. The idle tasks on each core have the same responsibilities as their vanilla counterparts.

Scheduler Suspension  Vanilla FreeRTOS allows the scheduler to be suspended/resumed by calling vTaskSuspendAll() and xTaskResumeAll() respectively. While the scheduler is suspended:

• Task switching is disabled but interrupts are left enabled.
• Calling any blocking/yielding function is forbidden, and time slicing is disabled.
• The tick count is frozen, but the tick interrupt still occurs to execute the application tick hook.

On scheduler resumption, xTaskResumeAll() catches up all of the lost ticks and unblock any timed-out tasks.

In ESP-IDF FreeRTOS, suspending the scheduler across multiple cores is not possible. Therefore when vTaskSuspendAll() is called on a particular core (e.g., core A):

• Task switching is disabled only on core A but interrupts for core A are left enabled.
• Calling any blocking/yielding function on core A is forbidden. Time slicing is disabled on core A.
• If an interrupt on core A unblocks any tasks, tasks with affinity to core A will go into core A’s own pending ready task list. Unpinned tasks or tasks with affinity to other cores can be scheduled on cores with the scheduler running.
• If the scheduler is suspended on all cores, tasks unblocked by an interrupt will be directed to the pending ready task lists of their pinned cores. For unpinned tasks, they will be placed in the pending ready list of the core where the interrupt occurred.
• If core A is CPU0, the tick count is frozen, and a pended tick count is incremented instead. However, the tick interrupt will still occur in order to execute the application tick hook.

When `xTaskResumeAll()` is called on a particular core (e.g., core A):

• Any tasks added to core A’s pending ready task list will be resumed.
• If core A is CPU0, the pended tick count is unwound to catch up with the lost ticks.

**Warning:** Given that scheduler suspension on ESP-IDF FreeRTOS only suspends scheduling on a particular core, scheduler suspension is **NOT** a valid method ensuring mutual exclusion between tasks when accessing shared data. Users should use proper locking primitives such as mutexes or spinlocks if they require mutual exclusion.

**Disabling Interrupts**  Vanilla FreeRTOS allows interrupts to be disabled and enabled by calling `taskDISABLE_INTERRUPTS` and `taskENABLE_INTERRUPTS` respectively.

ESP-IDF FreeRTOS provides the same API, however, interrupts are only disabled or enabled on the current core.

**Warning:** Disabling interrupts is a valid method of achieving mutual exclusion in Vanilla FreeRTOS (and single-core systems in general). However, in an SMP system, disabling interrupts is **NOT** a valid method ensuring mutual exclusion. Refer to Critical Sections for more details.

**Critical Sections**

**API Changes**  Vanilla FreeRTOS implements critical sections by disabling interrupts, which prevents preemptive context switches and the servicing of ISRs during a critical section. Thus a task/ISR that enters a critical section is guaranteed to be the sole entity to access a shared resource. Critical sections in Vanilla FreeRTOS have the following API:

• `taskENTER_CRITICAL()` enters a critical section by disabling interrupts
• `taskEXIT_CRITICAL()` exits a critical section by reenabling interrupts
• `taskENTER_CRITICAL_FROM_ISR()` enters a critical section from an ISR by disabling interrupt nesting
• `taskEXIT_CRITICAL_FROM_ISR()` exits a critical section from an ISR by reenabling interrupt nesting

However, in an SMP system, merely disabling interrupts does not constitute a critical section as the presence of other cores means that a shared resource can still be concurrently accessed. Therefore, critical sections in ESP-IDF FreeRTOS are implemented using spinlocks. To accommodate the spinlocks, the ESP-IDF FreeRTOS critical section APIs contain an additional spinlock parameter as shown below:

• Spinlocks are of `portMUX_TYPE` (**not to be confused to FreeRTOS mutexes**)
• `taskENTER_CRITICAL(&spinlock)` enters a critical from a task context
• `taskEXIT_CRITICAL(&spinlock)` exits a critical section from a task context
• `taskENTER_CRITICAL_ISR(&spinlock)` enters a critical section from an interrupt context
• `taskEXIT_CRITICAL_ISR(&spinlock)` exits a critical section from an interrupt context

**Note:** The critical section API can be called recursively, i.e., nested critical sections. Entering a critical section multiple times recursively is valid so long as the critical section is exited the same number of times it was entered. However, given that critical sections can target different spinlocks, users should take care to avoid deadlocking when entering critical sections recursively.

Spinlocks can be allocated statically or dynamically. As such, macros are provided for both static and dynamic initialization of spinlocks, as demonstrated by the following code snippets.

• Allocating a static spinlock and initializing it using `portMUX_INITIALIZER_UNLOCKED`:
// Statically allocate and initialize the spinlock
static portMUX_TYPE my_spinlock = portMUX_INITIALIZER_UNLOCKED;

void some_function(void)
{
    taskENTER_CRITICAL(&my_spinlock);
    // We are now in a critical section
    taskEXIT_CRITICAL(&my_spinlock);
}

• Allocating a dynamic spinlock and initializing it using portMUX_INITIALIZER():

    // Allocate the spinlock dynamically
    portMUX_TYPE *my_spinlock = malloc(sizeof(portMUX_TYPE));
    // Initialize the spinlock dynamically
    portMUX_INITIALIZE(my_spinlock);
    ...
    taskENTER_CRITICAL(my_spinlock);
    // Access the resource
    taskEXIT_CRITICAL(my_spinlock);

Implementation In ESP-IDF FreeRTOS, the process of a particular core entering and exiting a critical section is as follows:

• For taskENTER_CRITICAL(&spinlock) or taskENTER_CRITICAL_ISR(&spinlock)
  1. The core disables its interrupts or interrupt nesting up to config-
     MAX_SYSCALL_INTERRUPT_PRIORITY.
  2. The core then spins on the spinlock using an atomic compare-and-set instruction until it acquires the lock.
     A lock is acquired when the core is able to set the lock’s owner value to the core’s ID.
  3. Once the spinlock is acquired, the function returns. The remainder of the critical section runs with
     interrupts or interrupt nesting disabled.
• For taskEXIT_CRITICAL(&spinlock) or taskEXIT_CRITICAL_ISR(&spinlock)
  1. The core releases the spinlock by clearing the spinlock’s owner value.
  2. The core re-enables interrupts or interrupt nesting.

Restrictions and Considerations Given that interrupts (or interrupt nesting) are disabled during a critical section,
there are multiple restrictions regarding what can be done within critical sections. During a critical section, users
should keep the following restrictions and considerations in mind:

• Critical sections should be kept as short as possible
  – The longer the critical section lasts, the longer a pending interrupt can be delayed.
  – A typical critical section should only access a few data structures and/or hardware registers.
  – If possible, defer as much processing and/or event handling to the outside of critical sections.
• FreeRTOS API should not be called from within a critical section
• Users should never call any blocking or yielding functions within a critical section

Misc

ESP-IDF FreeRTOS Single Core Although ESP-IDF FreeRTOS is an SMP scheduler, some ESP targets are
single-core (such as ESP32-S2 and ESP32-C3). When building ESP-IDF applications for these targets, ESP-IDF
FreeRTOS is still used but the number of cores will be set to 1 (i.e., the CONFIG_FREERTOS_UNICORE will always
be enabled for single-core targets).

For multicore targets (such as ESP32 and ESP32-S3), CONFIG_FREERTOS_UNICORE can also be set. This results
in ESP-IDF FreeRTOS only running on CPU0, and all other cores will be inactive.
Note: Users should bear in mind that enabling `CONFIG_FREERTOS_UNICORE` is NOT equivalent to running Vanilla FreeRTOS. The additional APIs of ESP-IDF FreeRTOS can still be called, and the behavior changes of ESP-IDF FreeRTOS incur a small amount of overhead even when compiled for only a single core.

API Reference

This section introduces FreeRTOS types, functions, and macros. It is automatically generated from FreeRTOS header files.

Task API

Header File

- `components/freertos/FreeRTOS-Kernel/include/freertos/task.h`
- This header file can be included with:

```
#include "freertos/task.h"
```

Functions

static inline BaseType_t xTaskCreate(TaskFunction_t pxTaskCode, const char*const pcName, const configSTACK_DEPTH_TYPE usStackDepth, void*const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pxCreatedTask)

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 https://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:

```c
// 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;
}
```

(continues on next page)
Create the task, storing the handle. Note that the passed parameter ucParameterToPass must exist for the lifetime of the task, so in this case it is declared static. If it was just 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.

```c
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY, &xHandle );
configASSERT( xHandle );
```

Use the handle to delete the task.

```c
if( xHandle != NULL )
{
    vTaskDelete( xHandle );
}
```

### Parameters
- **pxTaskCode** - 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 )`.
- **pxCreatedTask** - Used to pass back a handle by which the created task can be referenced.

### Returns
pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

```c
static inline TaskHandle_t xTaskCreateStatic(TaskFunction_t pxTaskCode, 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 [https://www.FreeRTOS.org/a00111.html](https://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:

```c
// Dimensions of the buffer that the task being created will use as its stack.
// NOTE: This is the number of words the stack will hold, not the number of
// bytes. For example, if each stack item is 32-bits, and this is set to
// 100,
// then 400 bytes (100 * 32-bits) will be allocated.
#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];

// 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 words, not bytes.
        ( void * ) 1, // Parameter passed into the task.
        tskIDLE_PRIORITY, // Priority at which the task is created.
        &xTaskBuffer ); // Variable to hold the task's data

    // 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 );
}
```

**Note:** If `configNUMBER_OF_CORES > 1`, this function will create an unpinned task (see `tskNO_AFFINITY` for more details).

**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

- **pxTaskCode** - 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.
- **puxStackBuffer** - 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.

**Returns** If neither puxStackBuffer nor pxTaskBuffer are NULL, then the task will be created and a handle to the created task is returned. If either puxStackBuffer or pxTaskBuffer are NULL then the task will not be created and NULL is returned.

```c
void vTaskAllocateMPURegions(TaskHandle_t xTask, const MemoryRegion_t*const pxRegions)
```

Memory regions are assigned to a restricted task when the task is created by a call to xTaskCreateRestricted(). These regions can be redefined using vTaskAllocateMPURegions().

Example usage:

```c
// Define an array of MemoryRegion_t structures that configures an MPU region // allowing read/write access for 1024 bytes starting at the beginning of the // ucOneKByte array. The other two of the maximum 3 definable regions are // unused so set to zero.
static const MemoryRegion_t xAltRegions[ portNUM_CONFIGURABLE_REGIONS ] =
{
    // Base address Length Parameters
    { ucOneKByte, 1024, portMPU_REGION_READ_WRITE },
    { 0, 0, 0 },
    { 0, 0, 0 },
};

void vATask(void *pvParameters)
{
    // This task was created such that it has access to certain regions of // memory as defined by the MPU configuration. At some point it is // desired that these MPU regions are replaced with that defined in the // xAltRegions const struct above. Use a call to vTaskAllocateMPURegions() // for this purpose. NULL is used as the task handle to indicate that this // function should modify the MPU regions of the calling task.
    vTaskAllocateMPURegions( NULL, xAltRegions );

    // Now the task can continue its function, but from this point on can only // access its stack and the ucOneKByte array (unless any other statically // defined or shared regions have been declared elsewhere).
}
```

Parameters

- **xTask** - The handle of the task being updated.
- **pxRegions** - A pointer to a MemoryRegion_t structure that contains the new memory region definitions.
void vTaskDelete(TaskHandle_t xTaskToDelete)

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

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.

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.

See the demo application file death.c for sample code that utilises vTaskDelete().

Example usage:

```c
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 affect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes. See xTaskDelayUntil() 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:

```c
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 );
    }

    (continues on next page)
```
Parameters `xTicksToDelay` - The amount of time, in tick periods, that the calling task should block.

```c
BaseType_t xTaskDelayUntil (TickType_t *const pxPreviousWakeTime, const TickType_t xTimeIncrement)
```

INCLUDE_xTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

Delay a task until a specified time. 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, `xTaskDelayUntil()` specifies the absolute (exact) time at which it wishes to unblock.

The macro `pdMS_TO_TICKS()` can be used to calculate the number of ticks from a time specified in milliseconds with a resolution of one tick period.

Example usage:

```c
// Perform an action every 10 ticks.
vTaskFunction( void * pvParameters )
{
    TickType_t xLastWakeTime;
    const TickType_t xFrequency = 10;
    BaseType_t xWasDelayed;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();
    for( ;; )
    {
        // Wait for the next cycle.
        xWasDelayed = xTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here. xWasDelayed value can be used to determine
        // whether a deadline was missed if the code here took too long.
    }
}
```

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 `xTaskDelayUntil()`.

- `xTimeIncrement` - The cycle time period. The task will be unblocked at time `*pxPreviousWakeTime + xTimeIncrement`. Calling `xTaskDelayUntil` with the same `xTimeIncrement` parameter value will cause the task to execute with a fixed interface period.

Returns Value which can be used to check whether the task was actually delayed. Will be `pdTRUE` if the task was delayed and `pdFALSE` otherwise. A task will not be delayed if the next expected wake time is in the past.
**Chapter 2. API Reference**

**BaseType_t xTaskAbortDelay (TaskHandle_t xTask)**

INCLUDE_xTaskAbortDelay must be defined as 1 in FreeRTOSConfig.h for this function to be available.

A task will enter the Blocked state when it is waiting for an event. The event it is waiting for can be a temporal event (waiting for a time), such as when vTaskDelay() is called, or an event on an object, such as when xQueueReceive() or ulTaskNotifyTake() is called. If the handle of a task that is in the Blocked state is used in a call to xTaskAbortDelay() then the task will leave the Blocked state, and return from whichever function call placed the task into the Blocked state.

There is no ‘FromISR’ version of this function as an interrupt would need to know which object a task was blocked on in order to know which actions to take. For example, if the task was blocked on a queue the interrupt handler would then need to know if the queue was locked.

**Parameters**
- **xTask** – The handle of the task to remove from the Blocked state.

**Returns**
- If the task referenced by xTask was not in the Blocked state then pdFAIL is returned. Otherwise pdPASS is returned.

**Example usage:**

```c
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 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.
    }
}
```

**Parameters**
- **xTask** – Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

**Returns**
- The priority of xTask.

**UBaseType_t uxTaskPriorityGetFromISR (const TaskHandle_t xTask)**

A version of uxTaskPriorityGet() that can be used from an ISR.
**eTaskState** `eTaskGetState (TaskHandle_t xTask)`

INCLUDE `eTaskGetState` must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the state of any task. States are encoded by the `eTaskState` enumerated type.

**Parameters**
- **xTask** – Handle of the task to be queried.

**Returns**
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.

void `vTaskGetInfo (TaskHandle_t xTask, TaskStatus_t *pxTaskStatus, BaseType_t xGetFreeStackSpace, eTaskState eState)`

 configUSE_TRACE_FACILITY must be defined as 1 for this function to be available. See the configuration section for more information.

Populates a TaskStatus_t structure with information about a task.

Example usage:

```c
void vAFunction( void )
{
    TaskHandle_t xHandle;
    TaskStatus_t xTaskDetails;

    // Obtain the handle of a task from its name.
    xHandle = xTaskGetHandle("Task_Name");

    // Check the handle is not NULL.
    configASSERT( xHandle );

    // Use the handle to obtain further information about the task.
    vTaskGetInfo( xHandle,
                 &xTaskDetails,
                 pdTRUE, // Include the high water mark in xTaskDetails.
                 eInvalid ); // Include the task state in xTaskDetails.
}
```

**Parameters**
- **xTask** – Handle of the task being queried. If `xTask` is NULL then information will be returned about the calling task.
- **pxTaskStatus** – A pointer to the TaskStatus_t structure that will be filled with information about the task referenced by the handle passed using the `xTask` parameter.
- **xGetFreeStackSpace** – The TaskStatus_t structure contains a member to report the stack high water mark of the task being queried. Calculating the stack high water mark takes a relatively long time, and can make the system temporarily unresponsive - so the `xGetFreeStackSpace` parameter is provided to allow the high water mark checking to be skipped. The high watermark value will only be written to the TaskStatus_t structure if `xGetFreeStackSpace` is not set to `pdFALSE`;
- **eState** – The TaskStatus_t structure contains a member to report the state of the task being queried. Obtaining the task state is not as fast as a simple assignment - so the `eState` parameter is provided to allow the state information to be omitted from the TaskStatus_t structure. To obtain state information then set `eState` to `eInvalid` - otherwise the value passed in `eState` will be reported as the task state in the TaskStatus_t structure.

void `vTaskPrioritySet (TaskHandle_t xTask, UBaseType_t uxNewPriority)`

INCLUDE `vTaskPrioritySet` must be defined as 1 for this function to be available. See the configuration section for more information.

Set the priority of any task.
A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Example usage:

```c
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 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)`

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Suspend any task. 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:

```c
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)

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Resumes a suspended task.

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:

```c
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)

INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

An implementation of vTaskResume() that can be called from within an ISR.
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.

**Parameters**
- xTaskToResume – Handle to the task being readied.

**Returns**
- 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.

```c
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, xTaskDelayUntil(), xQueueSend(), etc.) must not be called while the scheduler is suspended.

Example usage:

```c
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();
    }
}
```
```c
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.
    
    // ...
    
    // 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();
    }
}
```

**Returns**  If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

**TickType_t xTaskGetTickCount (void)**

**Returns**  The count of ticks since vTaskStartScheduler was called.

**TickType_t xTaskGetTickCountFromISR (void)**

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.

**Returns**  The count of ticks since vTaskStartScheduler was called.

**UBaseType_t uxTaskGetNumberOfTasks (void)**

**Returns**  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 *pcTaskGetName (TaskHandle_t xTaskToQuery)**

**Returns**  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.

**TaskHandle_t xTaskGetHandle (const char *pcNameToQuery)**

**NOTE:** This function takes a relatively long time to complete and should be used sparingly.
Returns The handle of the task that has the human readable name pcNameToQuery. NULL is returned if no matching name is found. INCLUDE_xTaskGetHandle must be set to 1 in FreeRTOSConfig.h for pcTaskGetHandle() to be available.

BaseType_t xTaskGetStaticBuffers (TaskHandle_t xTask, StackType_t **ppuxStackBuffer, StaticTask_t **ppxTaskBuffer)

Retrieve pointers to a statically created task’s data structure buffer and stack buffer. These are the same buffers that are supplied at the time of creation.

Parameters
- xTask – The task for which to retrieve the buffers.
- ppuxStackBuffer – Used to return a pointer to the task’s stack buffer.
- ppxTaskBuffer – Used to return a pointer to the task’s data structure buffer.

Returns pdTRUE if buffers were retrieved, pdFALSE otherwise.

UBaseType_t uxTaskGetStackHighWaterMark (TaskHandle_t xTask)

Returns the high water mark of the stack associated with xTask. That is, the minimum free stack space there has been (in words, so on a 32 bit machine a value of 1 means 4 bytes) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

uxTaskGetStackHighWaterMark() and uxTaskGetStackHighWaterMark2() are the same except for their return type. Using configSTACK_DEPTH_TYPE allows the user to determine the return type. It gets around the problem of the value overflowing on 8-bit types without breaking backward compatibility for applications that expect an 8-bit return type.

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.

Returns The smallest amount of free stack space there has been (in words, so actual spaces on the stack rather than bytes) since the task referenced by xTask was created.

configSTACK_DEPTH_TYPE uxTaskGetStackHighWaterMark2 (TaskHandle_t xTask)

Returns the high water mark of the stack associated with xTask. That is, the minimum free stack space there has been (in words, so on a 32 bit machine a value of 1 means 4 bytes) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

uxTaskGetStackHighWaterMark() and uxTaskGetStackHighWaterMark2() are the same except for their return type. Using configSTACK_DEPTH_TYPE allows the user to determine the return type. It gets around the problem of the value overflowing on 8-bit types without breaking backward compatibility for applications that expect an 8-bit return type.

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.

Returns The smallest amount of free stack space there has been (in words, so actual spaces on the stack rather than bytes) since the task referenced by xTask was created.

void vTaskSetApplicationTaskTag (TaskHandle_t xTask, TaskHookFunction_t pxHookFunction)

Sets pxHookFunction to be the task hook function used by the task xTask. Passing xTask as NULL has the effect of setting the calling task’s hook function.

TaskHookFunction_t xTaskGetApplicationTaskTag (TaskHandle_t xTask)

Returns the pxHookFunction value assigned to the task xTask. Do not call from an interrupt service routine - call xTaskGetApplicationTaskTagFromISR() instead.

TaskHookFunction_t xTaskGetApplicationTaskTagFromISR (TaskHandle_t xTask)

Returns the pxHookFunction value assigned to the task xTask. Can be called from an interrupt service routine.
void vTaskSetThreadLocalStoragePointer (TaskHandle_t xTaskToSet, BaseType_t xIndex, void *pvValue)

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. The following two functions are used to set and query a pointer respectively.

void *pvTaskGetThreadLocalStoragePointer (TaskHandle_t xTaskToQuery, BaseType_t xIndex)

void vApplicationGetIdleTaskMemory (StaticTask_t **ppxIdleTaskTCBBuffer, StackType_t **ppxIdleTaskStackBuffer, uint32_t *pulIdleTaskStackSize)

This function is used to provide a statically allocated block of memory to FreeRTOS to hold the Idle Task TCB. This function is required when configSUPPORT_STATIC_ALLOCATION is set. For more information see this URI: https://www.FreeRTOS.org/a00110.html#configSUPPORT_STATIC_ALLOCATION

Parameters
- **ppxIdleTaskTCBBuffer** — A handle to a statically allocated TCB buffer
- **ppxIdleTaskStackBuffer** — A handle to a statically allocated Stack buffer for the idle task
- **pulIdleTaskStackSize** — A pointer to the number of elements that will fit in the allocated stack buffer

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.

pvParameter is 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)

xTaskGetIdleTaskHandle() is only available if INCLUDE_xTaskGetIdleTaskHandle is set to 1 in FreeRTOSConfig.h.

Simply returns the handle of the idle task of the current core. It is not valid to call xTaskGetIdleTaskHandle() before the scheduler has been started.

UBaseType_t uxTaskGetSystemState (TaskStatus_t *const pxTaskStatusArray, const UBaseType_t uxArraySize, configRUN_TIME_COUNTER_TYPE *const pulTotalRunTime)

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.

NOTE: This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

Example usage:

```c
// 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;
    configRUN_TIME_COUNTER_TYPE ulTotalRunTime, ulStatsAsPercentage;
```

(continues on next page)
// Make sure the write buffer does not contain a string.
pxWriteBuffer = 0x00;

// Take a snapshot of the number of tasks in case it changes while this
// function is executing.
uxArraySize = uxTaskGetNumberOfTasks();

// 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\t%lu\t\t%lu%%\n",
                        pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter,
                        ulStatsAsPercentage);
            }
            else
            {
                sprintf( pcWriteBuffer, "%s\t\t\t<1%%\n",
                        pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter);
            }

            pcWriteBuffer += strlen((char*)pcWriteBuffer);
        }
    }

    // The array is no longer needed, free the memory it consumes.
    vPortFree( pxTaskStatusArray );
}

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.
Chapter 2. API Reference

- **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 https://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.

**Returns** - 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.

```c
void vTaskList (char *pcWriteBuffer)
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.

NOTE 1: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Lists all the current tasks, along with their current state and stack usage high water mark.

Tasks are reported as blocked (‘B’), ready (‘R’), deleted (‘D’) or suspended (‘S’).

PLEASE 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() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays task: names, states, priority, stack usage and task number. Stack usage specified as the number of unused StackType_t words stack can hold on top of stack - not the number of bytes.

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.

```c
void vTaskGetRunTimeStats (char *pcWriteBuffer)
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.

NOTE 1: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

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 2:

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() 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.

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.

configRUN_TIME_COUNTER_TYPE ulTaskGetIdleRunTimeCounter (void)

configGENERATE_RUN_TIME_STATS, configUSE_STATS_FORMATTING_FUNCTIONS and INCLUDE_xTaskGetIdleTaskHandle must all be defined as 1 for these functions 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. While uxTaskGetSystemState() and vTaskGetRunTimeStats() writes the total execution time of each task into a buffer, ulTaskGetIdleRunTimeCounter() returns the total execution time of just the idle task and ulTaskGetIdleRunTimePercent() returns the percentage of the CPU time used by just the idle task.

Note: If configNUMBER_OF_CORES > 1, calling this function will query the idle task of the current core.

Returns The total run time of the idle task or the percentage of the total run time consumed by the idle task. This is the amount of time the idle task has actually been executing. The unit of time is dependent on the frequency configured using the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() macros.

configRUN_TIME_COUNTER_TYPE ulTaskGetIdleRunTimePercent (void)

 BaseType_t xTaskGenericNotifyWait (UBaseType_t uxIndexToWaitOn, uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t *pulNotificationValue, TickType_t xTicksToWait)

Waits for a direct to task notification to be pending at a given index within an array of direct to task notifications. See https://www.FreeRTOS.org/RTOS-task-notifications.html for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

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 one of the task’s notification values. 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 xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). 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 xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyWait() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotifyWait() is equivalent to calling xTaskNotifyWaitIndexed() with the uxIndexToWaitOn parameter set to 0.

**Parameters**

- **uxIndexToWaitOn** – The index within the calling task’s array of notification values on which the calling task will wait for a notification to be received. uxIndexToWaitOn must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyWait() does not have this parameter and always waits for notifications on index 0.

- **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_TICKS( value_in_ms ) can be used to convert a time specified in milliseconds to a time specified in ticks.

**Returns** If a notification was received (including notifications that were already pending when xTaskNotifyWait was called) then pdPASS is returned. Otherwise pdFAIL is returned.

```c
void vTaskGenericNotifyGiveFromISR( TaskHandle_t xTaskToNotify, UBaseType_t uxIndexToNotify, BaseType_t *pxHigherPriorityTaskWoken )
```

A version of xTaskNotifyGiveIndexed() that can be called from an interrupt service routine (ISR).

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this macro to be available.

Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

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 one of the task’s notification values. 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.

vTaskNotifyGiveIndexedFromISR() 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 vTaskNotifyGiveIndexedFromISR().

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 uTaskNotifyTakeIndexed() API function rather than the xTaskNotifyWaitIndexed() API function.

NOTE Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyFromISR() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyGiveFromISR() is equivalent to calling xTaskNotifyGiveIndexedFromISR() with the uxIndexToNotify parameter set to 0.

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().
- **uxIndexToNotify** - The index within the target task’s array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyGiveFromISR() does not have this parameter and always sends notifications to index 0.
- **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 is dependent on the port - see the documentation page for the port in use.

```c
BaseType_t xTaskGenericNotifyStateClear(TaskHandle_t xTask, UBaseType_t uxIndexToClear)
```


configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

If a notification is sent to an index within the array of notifications then the notification at that index is said to be ‘pending’ until it is read or explicitly cleared by the receiving task. xTaskNotifyStateClearIndexed() is the function that clears a pending notification without reading the notification value. The notification value at the same array index is not altered. Set xTask to NULL to clear the notification state of the calling task.
Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyStateClear() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyStateClear() is equivalent to calling xTaskNotifyStateClearIndexed() with the uxIndexToNotify parameter set to 0.

Parameters

- **xTask** – The handle of the RTOS task that will have a notification state cleared. Set xTask to NULL to clear a notification state in the calling task. To obtain a task’s handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task’s name in a call to xTaskGetHandle().

- **uxIndexToClear** – The index within the target task’s array of notification values to act upon. For example, setting uxIndexToClear to 1 will clear the state of the notification at index 1 within the array. uxIndexToClear must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. ulTaskNotifyStateClear() does not have this parameter and always acts on the notification at index 0.

Returns pdTRUE if the task’s notification state was set to eNotWaitingNotification, otherwise pdFALSE.

```c
uint32_t ulTaskGenericNotifyValueClear (TaskHandle_t xTask, UBaseType_t uxIndexToClear, uint32_t ulBitsToClear)
```


Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

ulTaskNotifyValueClearIndexed() clears the bits specified by the ulBitsToClear bit mask in the notification value at array index uxIndexToClear of the task referenced by xTask.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. ulTaskNotifyValueClear() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling ulTaskNotifyValueClear() is equivalent to calling ulTaskNotifyValueClearIndexed() with the uxIndexToClear parameter set to 0.

Parameters

- **xTask** – The handle of the RTOS task that will have bits in one of its notification values cleared. Set xTask to NULL to clear bits in a notification value in the calling task. To obtain a task’s handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task’s name in a call to xTaskGetHandle().

- **uxIndexToClear** – The index within the target task’s array of notification values in which to clear the bits. uxIndexToClear must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. ulTaskNotifyValueClear() does not have this parameter and always clears bits in the notification value at index 0.

- **ulBitsToClear** – Bit mask of the bits to clear in the notification value of xTask. Set a bit to 1 to clear the corresponding bits in the task’s notification value. Set ulBitsToClear to 0xffffffff (UINT_MAX on 32-bit architectures) to clear the notification value to 0. Set ulBitsToClear to 0 to query the task’s notification value without clearing any bits.

Returns The value of the target task’s notification value before the bits specified by ulBitsToClear were cleared.

```c
void vTaskSetTimeOutState (TimeOut_t *const pxTimeOut)
```

Capture the current time for future use with xTaskCheckForTimeOut().
Parameters **pxTimeOut**—Pointer to a timeout object into which the current time is to be captured. The captured time includes the tick count and the number of times the tick count has overflowed since the system first booted.

BaseType_t **vTaskCheckForTimeOut** (TimeOut_t *const pxTimeOut, TickType_t *const pxTicksToWait)

Determines if pxTicksToWait ticks has passed since a time was captured using a call to vTaskSetTimeOut-State(). The captured time includes the tick count and the number of times the tick count has overflowed.

Example Usage:

```c
// Driver library function used to receive uxWantedBytes from an Rx buffer
// that is filled by a UART interrupt. If there are not enough bytes in the
// Rx buffer then the task enters the Blocked state until it is notified that
// more data has been placed into the buffer. If there is still not enough
// data then the task re-enters the Blocked state, and xTaskCheckForTimeOut()
// is used to re-calculate the block time to ensure the total amount of time
// spent in the Blocked state does not exceed MAX_TIME_TO_WAIT. This
// continues until either the buffer contains at least uxWantedBytes bytes,
// or the total amount of time spent in the Blocked state reaches
// MAX_TIME_TO_WAIT — at which point the task reads however many bytes are
// available up to a maximum of uxWantedBytes.

size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes )
{
    size_t uxReceived = 0;
    TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
    TimeOut_t xTimeOut;

    // Initialize xTimeOut. This records the time at which this function
    // was entered.
    vTaskSetTimeOutState( &xTimeOut );

    // Loop until the buffer contains the wanted number of bytes, or a
    // timeout occurs.
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
    {
        // The buffer didn’t contain enough data so this task is going to
        // enter the Blocked state. Adjusting xTicksToWait to account for
        // any time that has been spent in the Blocked state within this
        // function so far to ensure the total amount of time spent in the
        // Blocked state does not exceed MAX_TIME_TO_WAIT.
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
        {
            // Timed out before the wanted number of bytes were available,
            // exit the loop.
            break;
        }

        // Wait for a maximum of xTicksToWait ticks to be notified that the
        // receive interrupt has placed more data into the buffer.
        ulTaskNotifyTake( pdTRUE, xTicksToWait );
    }

    // Attempt to read uxWantedBytes from the receive buffer into pucBuffer.
    // The actual number of bytes read (which might be less than
    // uxWantedBytes) is returned.
    uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

    return uxReceived;
}
```

(continues on next page)
See also:
https://www.FreeRTOS.org/xTaskCheckForTimeOut.html

Parameters

- **pxTimeOut** - The time status as captured previously using vTaskSetTimeOutState. If the timeout has not yet occurred, it is updated to reflect the current time status.
- **pxTicksToWait** - The number of ticks to check for timeout i.e. if pxTicksToWait ticks have passed since pxTimeOut was last updated (either by vTaskSetTimeOutState() or xTaskCheckForTimeOut()), the timeout has occurred. If the timeout has not occurred, pxTicksToWait is updated to reflect the number of remaining ticks.

Returns
If timeout has occurred, pdTRUE is returned. Otherwise pdFALSE is returned and pxTicksToWait is updated to reflect the number of remaining ticks.

BaseType_t xTaskCatchUpTicks (TickType_t xTicksToCatchUp)
This function corrects the tick count value after the application code has held interrupts disabled for an extended period resulting in tick interrupts having been missed.

This function is similar to vTaskStepTick(), however, unlike vTaskStepTick(), xTaskCatchUpTicks() may move the tick count forward past a time at which a task should be removed from the blocked state. That means tasks may have to be removed from the blocked state as the tick count is moved.

Parameters **xTicksToCatchUp** - The number of tick interrupts that have been missed due to interrupts being disabled. Its value is not computed automatically, so must be computed by the application writer.

Returns pdTRUE if moving the tick count forward resulted in a task leaving the blocked state and a context switch being performed. Otherwise pdFALSE.

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.

**configRUN_TIME_COUNTER_TYPE ulRunTimeCounter**
The total run time allocated to the task so far, as defined by the run time stats clock. See [https://www.FreeRTOS.org/rtos-run-time-stats.html](https://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.

**configSTACK_DEPTH_TYPE 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 tskNO_AFFINITY). If configNUMBER_OF_CORES == 1, this will always be 0.

**Macros**

**tskIDLE_PRIORITY**
Defines the priority used by the idle task. This must not be modified.

**tskNO_AFFINITY**
Macro representing and unpinned (i.e., “no affinity”) task in xCoreID parameters

**taskVALID_CORE_ID (xCoreID)**
Macro to check if an xCoreID value is valid

    Returns  pdTRUE if valid, pdFALSE otherwise.

**taskYIELD ()**
Macro for forcing a context switch.

**taskENTER_CRITICAL (x)**
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_FROM_ISR ()**

**taskENTER_CRITICAL_ISR (x)**

**taskEXIT_CRITICAL (x)**
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_FROM_ISR (x)**

**taskEXIT_CRITICAL_ISR (x)**
**taskDISABLE_INTERRUPTS()**

Macro to disable all maskable interrupts.

**taskENABLE_INTERRUPTS()**

Macro to enable microcontroller interrupts.

**taskSCHEDULER_SUSPENDED**

Definitions returned by xTaskGetSchedulerState(). taskSCHEDULER_SUSPENDED is 0 to generate more optimal code when configASSERT() is defined as the constant is used in assert() statements.

**taskSCHEDULER_NOT_STARTED**

**taskSCHEDULER_RUNNING**

**xTaskNotifyIndexed** (xTaskToNotify, uxIndexToNotify, ulValue, eAction)


configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Sends a direct to task notification to a task, with an optional value and action.

Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

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 one of the task’s notification values. 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 task can use xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification to be pending. The task does not consume any CPU time while it is in the Blocked state.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). 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.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotify() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotify() is equivalent to calling xTaskNotifyIndexed() with the uxIndexToNotify parameter set to 0.

**eSetBits** - The target notification value is bitwise ORed with ulValue. xTaskNotifyIndexed() always returns pdPASS in this case.

**eIncrement** - The target notification value is incremented. ulValue is not used and xTaskNotifyIndexed() always returns pdPASS in this case.

**eSetValueWithOverwrite** - The target notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification at the same array index (the task already had a notification pending at that index). xTaskNotifyIndexed() always returns pdPASS in this case.
eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending at the same array index then the target notification value is set to ulValue and xTaskNotifyIndexed() will return pdPASS. If the task being notified already had a notification pending at the same array index then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification at the specified array index without the notification value at that index being updated. ulValue is not used and xTaskNotifyIndexed() always returns pdPASS in this case.

pulPreviousNotificationValue - Can be used to pass out the subject task’s notification value before any bits are modified by the notify function.

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().
- uxIndexToNotify - The index within the target task’s array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotify() does not have this parameter and always sends notifications to index 0.
- 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:

Returns 
Dependent on the value of eAction. See the description of the eAction parameter.

xTaskNotifyAndQueryIndexed (xTaskToNotify, uxIndexToNotify, ulValue, eAction, pulPreviousNotifyValue)


xTaskNotifyAndQueryIndexed() performs the same operation as xTaskNotifyIndexed() with the addition that it also returns the subject task’s prior notification value (the notification value at the time the function is called rather than when the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQuery() performs the same operation as xTaskNotify() with the addition that it also returns the subject task’s prior notification value (the notification value as it was at the time the function is called, rather than when the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyIndexedFromISR (xTaskToNotify, uxIndexToNotify, ulValue, eAction, pxHigherPriorityTaskWoken)


configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

A version of xTaskNotifyIndexed() that can be used from an interrupt service routine (ISR).

Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

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 one of the task’s notification values. 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 task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state
Chapter 2. API Reference

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.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyFromISR() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyFromISR() is equivalent to calling xTaskNotifyIndexedFromISR() with the uxIndexToNotify parameter set to 0.

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.

**Parameters**

- **uxIndexToNotify** — The index within the target task’s array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyFromISR() does not have this parameter and always sends notifications to index 0.
- **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:
  - **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.

**Returns** — Dependent on the value of eAction. See the description of the eAction parameter.

xTaskNotifyAndQueryIndexedFromISR(xTaskToNotify, uxIndexToNotify, ulValue, eAction, pulPreviousNotificationValue, pxHigherPriorityTaskWoken)


xTaskNotifyAndQueryIndexedFromISR() performs the same operation as xTaskNotifyIndexedFromISR() with the addition that it also returns the subject task’s prior notification value (the notification value at the time the function is called rather than at the time the function returns) in the additional pulPreviousNotificationValue parameter.

xTaskNotifyAndQueryFromISR() performs the same operation as xTaskNotifyFromISR() with the addition that it also returns the subject task’s prior notification value (the notification value at the time the function is
called rather than at the time the function returns) in the additional pulPreviousNotifyValue parameter.

\section*{Chapter 2. API Reference}

\begin{itemize}
\item \textbf{xTaskNotifyWait} (ulBitsToClearOnEntry, ulBitsToClearOnExit, pulNotificationValue, xTicksToWait)
\item \textbf{xTaskNotifyWaitIndexed} (uxIndexToWaitOn, ulBitsToClearOnEntry, ulBitsToClearOnExit, pulNotificationValue, xTicksToWait)
\item \textbf{xTaskNotifyGiveIndexed} (xTaskToNotify, uxIndexToNotify)
\end{itemize}

\begin{itemize}
\item \textbf{Send a direct task notification to a particular index in the target task’s notification array in a manner similar to giving a counting semaphore.}
\end{itemize}


\begin{itemize}
\item \textbf{configUSE_TASK_NOTIFICATIONS} must be undefined or defined as 1 for these macros to be available.
\end{itemize}

\begin{itemize}
\item Each task has a private array of “notification values” (or “notifications”), each of which is a 32-bit unsigned integer (uint32_t). The constant \textbf{configTASK_NOTIFICATION_ARRAY_ENTRIES} sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.
\end{itemize}

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 one of the task’s notification values. In that way task notifications can be used to send data to a task, or be used as lightweight and fast binary or counting semaphores.

\begin{itemize}
\item \textbf{xTaskNotifyGiveIndexed()} is a helper macro intended for use when task notifications are used as lightweight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given using the \textbf{xSemaphoreGive()} API function, the equivalent action that instead uses a task notification is \textbf{xTaskNotifyGiveIndexed()}. 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 \textbf{uTaskNotifyTakeIndexed()} API function rather than the \textbf{xTaskNotifyWaitIndexed()} API function.
\end{itemize}

\begin{itemize}
\item \textbf{NOTE} Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.
\end{itemize}

\begin{itemize}
\item Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. \textbf{xTaskNotifyGive()} is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling \textbf{xTaskNotifyGive()} is equivalent to calling \textbf{xTaskNotifyGiveIndexed()} with the \textbf{uxIndexToNotify} parameter set to 0.
\end{itemize}

\begin{itemize}
\item \textbf{Parameters}
\begin{itemize}
\item \textbf{xTaskToNotify} –The handle of the task being notified. The handle to a task can be returned from the \textbf{xTaskCreate()} API function used to create the task, and the handle of the currently running task can be obtained by calling \textbf{xTaskGetCurrentTaskHandle()}.
\item \textbf{uxIndexToNotify} –The index within the target task’s array of notification values to which the notification is to be sent. \textbf{uxIndexToNotify} must be less than \textbf{configTASK_NOTIFICATION_ARRAY_ENTRIES}. \textbf{xTaskNotifyGive()} does not have this parameter and always sends notifications to index 0.
\end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{Returns} \textbf{xTaskNotifyGive()} is a macro that calls \textbf{xTaskNotify()} with the \textbf{eAction} parameter set to \textbf{eIncrement} - so \textbf{pDPass} is always returned.
\end{itemize}

\begin{itemize}
\item \textbf{vTaskNotifyGiveFromISR} (xTaskToNotify, pxHigherPriorityTaskWoken)
\item \textbf{vTaskNotifyGiveIndexedFromISR} (xTaskToNotify, uxIndexToNotify, pxHigherPriorityTaskWoken)
\end{itemize}
**ulTaskNotifyTakeIndexed** (uxIndexToWaitOn, xClearCountOnExit, xTicksToWait)

Waits for a direct to task notification on a particular index in the calling task’s notification array in a manner similar to taking a counting semaphore.


configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

Each task has a private array of “notification values” (or ‘notifications’), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

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 one of the task’s notification values. 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.

ulTaskNotifyTakeIndexed() 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 ulTaskNotifyTakeIndexed().

When a task is using its notification value as a binary or counting semaphore other tasks should send notifications to it using the xTaskNotifyGiveIndexed() macro, or xTaskNotifyIndex() function with the eAction parameter set to eIncrement.

ulTaskNotifyTakeIndexed() can either clear the task’s notification value at the array index specified by the uxIndexToWaitOn parameter to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the notification value on exit, in which case the notification value acts like a counting semaphore.

A task can use ulTaskNotifyTakeIndexed() to [optionally] block to wait for a notification. The task does not consume any CPU time while it is in the Blocked state.

Where as xTaskNotifyWaitIndexed() will return when a notification is pending, ulTaskNotifyTakeIndexed() will return when the task’s notification value is not zero.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single “notification value”, and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. ulTaskNotifyTake() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling ulTaskNotifyTake() is equivalent to calling ulTaskNotifyTakeIndexed() with the uxIndexToWaitOn parameter set to 0.

**Parameters**

- **uxIndexToWaitOn** - The index within the calling task’s array of notification values on which the calling task will wait for a notification to be non-zero. uxIndexToWaitOn must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyTake() does not have this parameter and always waits for notifications on index 0.
- **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_TICKS** (value_in_ms) can be used to convert a time specified in milliseconds to a time specified in ticks.

**Returns** The task’s notification count before it is either cleared to zero or decremented (see the xClearCountOnExit parameter).

- `xTaskNotifyStateClear` (xTask)
- `xTaskNotifyStateClearIndexed` (xTask, uxIndexToClear)
- `ulTaskNotifyValueClear` (xTask, ulBitsToClear)
- `ulTaskNotifyValueClearIndexed` (xTask, uxIndexToClear, ulBitsToClear)

**Type Definitions**

- `typedef struct tskTaskControlBlock *TaskHandle_t`
- `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.

**Enumerations**

- `enum eTaskState`

  Task states returned by eTaskGetState.

  **Values:**

  - **enumerator eRunning**
    
    A task is querying the state of itself, so must be running.
  
    enumerator eReady
    
    The task being queried is in a ready or pending ready list.
  
    enumerator eBlocked
    
    The task being queried is in the Blocked state.
  
    enumerator eSuspended
    
    The task being queried is in the Suspended state, or is in the Blocked state with an infinite time out.
  
    enumerator eDeleted
    
    The task being queried has been deleted, but its TCB has not yet been freed.
  
    enumerator eInvalid
    
    Used as an ‘invalid state’ value.

- `enum eNotifyAction`

  Actions that can be performed when vTaskNotify() is called.

  **Values:**
enumerator **eNoAction**
   Notify the task without updating its notify value.

enumerator **eSetBits**
   Set bits in the task’s notification value.

enumerator **eIncrement**
   Increment the task’s notification value.

enumerator **eSetValueWithOverwrite**
   Set the task’s notification value to a specific value even if the previous value has not yet been read by the task.

enumerator **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:

enumerator **eAbortSleep**
   A task has been made ready or a context switch pended since portSUPPRESS_TICKS_AND_SLEEP() was called - abort entering a sleep mode.

enumerator **eStandardSleep**
   Enter a sleep mode that will not last any longer than the expected idle time.

Queue API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/queue.h
- This header file can be included with:

```
#include "freertos/queue.h"
```

Functions

```c
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)
{
    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.
}

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).

Returns

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

BaseType_t xQueuePeek(QueueHandle_t xQueue, void *const pvBuffer, TickType_t xTicksToWait)

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:

```c
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.
    }

    // ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // Peek a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueuePeek( xQueue, &pxRxedMessage, ( TickType_t ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask, but the item still remains on the queue.
        }
    }

    // ... Rest of task code.
}
```

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.

**Returns** pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

```
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().

**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.

**Returns** pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

```
BaseType_t xQueueReceive(QueueHandle_t xQueue, void *const pvBuffer, TickType_t xTicksToWait)
```
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:**

```c
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;
    (continues on next page)
```
xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

// ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &pxRxedMessage, ( TickType_t ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }

    // ... Rest of task code.
}

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.

Returns  pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue )

Return the number of messages stored in a queue.

Parameters xQueue - A handle to the queue being queried.

Returns  The number of messages available in the queue.

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.

Parameters xQueue - A handle to the queue being queried.

Returns  The number of spaces available in the queue.

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 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 xQueueSendTo-
BackFromISR() 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):

```c
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
    {
        // 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 )
    {
        portYIELD_FROM_ISR();
    }
}
```

**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.
- `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).

**Returns**
- `pdTRUE` if the data was successfully sent to the queue, otherwise `errQUEUE_FULL`.

```c
BaseType_t xQueueGiveFromISR(QueueHandle_t xQueue, BaseType_t *const pxHigherPriorityTaskWoken)
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:

```c
QueueHandle_t xQueue;
```
// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
    char cValueToPost;
    const TickType_t xTicksToWait = ( TickType_t )0xff;

    // Create a queue capable of containing 10 characters.
    xQueue = xQueueCreate( 10, sizeof( char ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ... Posting some characters ...
    cValueToPost = 'a';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
    cValueToPost = 'b';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );

    // ... keep posting characters ... this task may block when the queue
    // 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 cRxedChar;

    while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar, &
                                xTaskWokenByReceive ) )
    {
        // A character was received. Output the character now.
        vOutputCharacter( cRxedChar );

        // If removing the character from the queue woke the task that was
        // posting onto the queue xTaskWokenByReceive will have been set to
        // pdTRUE. No matter how many times this loop iterates only one
        // task will be woken.
    }

    if( xTaskWokenByReceive != ( char ) pdFALSE;
    {
        taskYIELD();
    }
}

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.
- **pxHigherPriorityTaskWoken** - A task may be blocked waiting for space to be-
  become available on the queue. If xQueueReceiveFromISR causes such a task to unblock
  *pxTaskWoken will get set to pdTRUE, otherwise *pxTaskWoken will remain unchanged.

Returns  pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.
BaseType_t xQueueIsQueueEmptyFromISR (const QueueHandle_t xQueue)
Queries a queue to determine if the queue is empty. This function should only be used in an ISR.

**Parameters**
- **xQueue** – The handle of the queue being queried

**Returns**
- pdFALSE if the queue is not empty, or pdTRUE if the queue is empty.

BaseType_t xQueueIsQueueFullFromISR (const QueueHandle_t xQueue)
Queries a queue to determine if the queue is full. This function should only be used in an ISR.

**Parameters**
- **xQueue** – The handle of the queue being queried

**Returns**
- pdFALSE if the queue is not full, or pdTRUE if the queue is full.

UBaseType_t uxQueueMessagesWaitingFromISR (const QueueHandle_t xQueue)
A version of uxQueueMessagesWaiting() that can be called from an ISR. Return the number of messages stored in a queue.

**Parameters**
- **xQueue** – A handle to the queue being queried.

**Returns**
- The number of messages available in the queue.

void vQueueAddToRegistry (QueueHandle_t xQueue, const char* pcQueueName)
The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call vQueueAddToRegistry() to 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 affect the number of queues, semaphores and mutexes that can be created - just the number that the registry can hold.

If vQueueAddToRegistry is called more than once with the same xQueue parameter, the registry will store the pcQueueName parameter from the most recent call to vQueueAddToRegistry.

**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.
- **pcQueueName** – 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() to 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.

**Parameters**
- **xQueue** – The handle of the queue the name of which will be returned.

**Returns**
- 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.

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 https://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.

**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.

**Returns** If the queue set is created successfully then a handle to the created queue set is returned. Otherwise NULL is returned.

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.

**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.

**Returns** 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.

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.

**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.
Returns  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.

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 https://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.

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.

Returns xQueueSelectFromSet() will return the handle of a queue (cast to a QueueSetMember-
Handle_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.

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, 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 https://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.


Example usage:

```c
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.

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.

Returns

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.

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 https://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.


Example usage:

```c
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
};

#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )

// ucQueueBuffer 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 )
{
    // ... Rest of task code.
}
```

(continues on next page)
QueueHandle_t xQueue1;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( QUEUE_LENGTH, // The number of items the queue can...
ITEM_SIZE // The size of each item in the queue
& ucQueueStorage[ 0 ] ), // The buffer that will...
& xQueueBuffer ); // The buffer that will hold the...

// 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.

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 pucQueueStorage 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 pucQueueStorage can be NULL.
- pxQueueBuffer – Must point to a variable of type StaticQueue_t, which will be used
  to hold the queue’s data structure.

Returns If the queue is created then a handle to the created queue is returned. If pxQueueBuffer
is NULL then NULL is returned.

xQueueGetStaticBuffers (xQueue, ppucQueueStorage, ppxStaticQueue)

Retrieve pointers to a statically created queue’s data structure buffer and storage area buffer. These are the
same buffers that are supplied at the time of creation.

Parameters

- xQueue – The queue for which to retrieve the buffers.
- ppucQueueStorage – Used to return a pointer to the queue’s storage area buffer.
- ppxStaticQueue – Used to return a pointer to the queue’s data structure buffer.

Returns pdTRUE if buffers were retrieved, pdFALSE otherwise.

xQueueSendToFront (xQueue, pItemToQueue, xTicksToWait)

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:

```c
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( 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.
}

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.

Returns
- pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

**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
{
    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.
}

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.

Returns

- **pdTRUE** if the item was successfully posted, otherwise **errQUEUE_FULL**.

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:

```c
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.
}
```

**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.

**Returns**

`pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

`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:

```c
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!
}

// ...
}
```

**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.

**Returns** `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.
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):

```c
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 )
    {
        taskYIELD();
    }
}
```

**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.
- `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 xQueueSendToFrontFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

**Returns**

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

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 )
    {
        taskYIELD();
    }
}

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.
• pxHigherPriorityTaskWoken – xQueueSendToBackFromISR() will set *px-
  HigherPriorityTaskWoken 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 xQueue-
  SendToBackFromISR() sets this value to pdTRUE then a context switch should be re-
  quested before the interrupt is exited.

Returns pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

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:

```c
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.
    xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}
```

(continues on next page)
```c
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.
    }
}
```

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.
- **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.

Returns

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.

**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();
    }
}

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.
- 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.

Returns

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

xQueueReset (xQueue)

Reset a queue back to its original empty state. The return value is now obsolete and is always set to pdPASS.

Type Definitions

typedef struct QueueDefinition *QueueHandle_t

typedef struct QueueDefinition *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 struct QueueDefinition *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

- components/freertos/FreeRTOS-Kernel/include/freertos/semphr.h
- This header file can be included with:

```c
#include "freertos/semphr.h"
```

Macros

- `semBINARY_SEMAPHORE_QUEUE_LENGTH`
- `semSEMAPHORE_QUEUE_ITEM_LENGTH`
- `semGIVE_BLOCK_TIME`

`vSemaphoreCreateBinary(xSemaphore)`

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! [https://www.FreeRTOS.org/RTOS-task-notifications.html](https://www.FreeRTOS.org/RTOS-task-notifications.html)

This old `vSemaphoreCreateBinary()` macro is now deprecated in favour of the `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’.

`Macro` that implements 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 we don’t want to actually store any data - we just want to know if the queue is empty or full.

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:

```c
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.
    vSemaphoreCreateBinary( xSemaphore );

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

Parameters

- `xSemaphore` - Handle to the created semaphore. Should be of type `SemaphoreHandle_t`. 
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! [https://www.FreeRTOS.org/RTOS-task-notifications.html](https://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 [https://www.FreeRTOS.org/a00111.html](https://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 semaphore must first be ‘given’ before it can be ‘taken’.

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 xSemaphoreCreateBinary().
    // 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.
    }
}
```

**Returns** Handle to the created semaphore, or NULL if the memory required to hold the semaphore’s data structures could not be allocated.

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! [https://www.FreeRTOS.org/RTOS-task-notifications.html](https://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 [https://www.FreeRTOS.org/a00111.html](https://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:

```c
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.
}
```

**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.

**Returns**

If the semaphore is created then a handle to the created semaphore is returned. If `pxSemaphoreBuffer` is NULL then NULL is returned.

**xSemaphoreTake(xSemaphore, xBlockTime)**

Macro to obtain a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

Example usage:

```c
SemaphoreHandle_t xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
    // Create the semaphore to guard a shared resource.
    xSemaphore = xSemaphoreCreateBinary();
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xSemaphore != NULL )
    {
        // 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.
            // ...
        }
    }
```

(continues on next page)
// 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.
}

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).

**Returns** pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

**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:

```c
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 )
    {
        // 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.
}
}

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 xBlock-
  Time.

Returns pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the
semaphore becoming available.

xSemaphoreGive(xSemaphore)

Macro to release a semaphore. The semaphore must have previously been created with a call to
xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using
xSemaphoreTake().

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 )
{
  // Create the semaphore to guard a shared resource.
xSemaphore = vSemaphoreCreateBinary();

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.
        }
    }
}

Parameters

• xSemaphore - A handle to the semaphore being released. This is the handle returned
  when the semaphore was created.

Returns 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.

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.

```c
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.
        }
    }
}
```

Parameters

- **xMutex** - A handle to the mutex being released, or ‘given’. This is the handle returned by `xSemaphoreCreateMutex()`;

Returns

`pdTRUE` if the semaphore was given.

**xSemaphoreGiveFromISR** (xSemaphore, pxHigherPriorityTaskWoken)

Macro to release a semaphore. The semaphore must have previously been created with a call to `xSemaphoreCreateBinary()` 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:
```c
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT 10
SemaphoreHandle_t xSemaphore = NULL;

// 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.
    }
}
```

**Parameters**
- **xSemaphore** - A handle to the semaphore being released. This is the handle returned when the semaphore was created.
- **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 \texttt{xSemaphoreGiveFromISR()} sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

\textbf{Returns} pdTRUE if the semaphore was successfully given, otherwise \texttt{errQUEUE\_FULL}.

\texttt{xSemaphoreTakeFromISR} (xSemaphore, pxHigherPriorityTaskWoken)

\textit{Macro} to take a semaphore from an ISR. The semaphore must have previously been created with a call to \texttt{xSemaphoreCreateBinary()} or \texttt{xSemaphoreCreateCounting()].

Mutex type semaphores (those created using a call to \texttt{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).

\textbf{Parameters}

* xSemaphore – A handle to the semaphore being taken. This is the handle returned when the semaphore was created.
* pxHigherPriorityTaskWoken – \texttt{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 \texttt{xSemaphoreTakeFromISR()} sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

\textbf{Returns} pdTRUE if the semaphore was successfully taken, otherwise pdFALSE

\texttt{xSemaphoreCreateMutex}()

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 \texttt{xSemaphoreCreateMutex()} then the required memory is automatically dynamically allocated inside the \texttt{xSemaphoreCreateMutex()} function. (see https://www.FreeRTOS.org/a00111.html). If a mutex is created using \texttt{xSemaphoreCreateMutexStatic()} then the application writer must provided the memory. \texttt{xSemaphoreCreateMutexStatic()} therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the \texttt{xSemaphoreTake()} and \texttt{xSemaphoreGive()} macros. The \texttt{xSemaphoreTakeRecursive()} and \texttt{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 \texttt{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.

\textbf{Example usage:}

```c
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.
    }
}  
```

(continues on next page)
The semaphore can now be used.

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 https://www.FreeRTOS.org/a00111.html). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided 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:

```c
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.
}
```

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.

Returns If the mutex was successfully created then a handle to the created mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

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 https://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:

```c
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.
    }
}
```

**Returns**

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 https://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:

```c
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.
    xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );

    // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
    // so there is no need to check it.
}
```

**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.

**Returns**

If the recursive mutex was successfully created then a handle to the created recursive mutex is returned. If `pxStaticSemaphore` was `NULL` then `NULL` is returned.

**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! [https://www.FreeRTOS.org/RTOS-task-notifications.html](https://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 [https://www.FreeRTOS.org/a00111.html](https://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:

```c
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.
        // The semaphore can now be used.
    }
}
```

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.

Returns Handle to the created semaphore. Null if the semaphore could not be 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! [https://www.FreeRTOS.org/RTOS-task-notifications.html](https://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 [https://www.FreeRTOS.org/a00111.html](https://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:

```c
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.
}
```

**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.

**Returns**

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.

```c
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.

```c
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.

```c
xSemaphoreGetMutexHolderFromISR( 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.

```c
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.
uxSemaphoreGetCountFromISR(xSemaphore)
semphr.h

```c
UBaseType_t uxSemaphoreGetCountFromISR(SemaphoreHandle_t xSemaphore);
```

If the semaphore is a counting semaphore then uxSemaphoreGetCountFromISR() returns its current count value. If the semaphore is a binary semaphore then uxSemaphoreGetCountFromISR() returns 1 if the semaphore is available, and 0 if the semaphore is not available.

xSemaphoreGetStaticBuffer(xSemaphore, ppxSemaphoreBuffer)

Retrieve pointer to a statically created binary semaphore, counting semaphore, or mutex semaphore’s data structure buffer. This is the same buffer that is supplied at the time of creation.

**Parameters**
- `xSemaphore` - The semaphore for which to retrieve the buffer.
- `pxSemaphoreBuffer` - Used to return a pointer to the semaphore’s data structure buffer.

**Returns**
- `pdTRUE` if buffer was retrieved, `pdFALSE` otherwise.

**Type Definitions**

typedef `QueueHandle_t` SemaphoreHandle_t

**Timer API**

**Header File**

- `components/freertos/FreeRTOS-Kernel/include/freertos/timers.h`
- This header file can be included with:

```c
#include "freertos/timers.h"
```

**Functions**

TimerHandle_t xTimerCreate(const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const BaseType_t xAutoReload, 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 https://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:**

```c
* #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 + 1 ) ), // 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 /* { */
    /*   // 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( ;; );
*

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 xTimerPeri-
odInTicks 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\_R\_H\_Z is less than or equal to 1000. Time timer period must be greater than 0.
- **xAutoReload** – If xAutoReload is set to pdTRUE then the timer will expire repeatedly with a frequency set by the xTimerPeriodInTicks parameter. If xAutoReload 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 );”.

Returns

If the timer is successfully created then a handle to the newly created timer is returned. If the timer cannot be created because there is insufficient FreeRTOS heap remaining to allocate the timer structures then NULL is returned.

```
TimerHandle_t xTimerCreateStatic
(const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const BaseType_t xAutoReload, 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 automatical-
dynamically allocated inside the xTimerCreate() function. (see https://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
Timers are created in the dormant state. The \texttt{xTimerStart()}, \texttt{xTimerReset()}, \texttt{xTimerStartFromISR()}, \texttt{xTimerResetFromISR()}, \texttt{xTimerChangePeriod()} and \texttt{xTimerChangePeriodFromISR()} API functions can all be used to transition a timer into the active state.

Example usage:

```c
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. \texttt{xTimerCreateStatic()} has an extra parameter than the normal \texttt{xTimerCreate()} API function. The parameter \texttt{is} a pointer to the StaticTimer_t structure that will hold the software timer structure. If the parameter \texttt{is} passed as NULL then the structure will be allocated dynamically, just as if \texttt{xTimerCreate()} had been called.
    xTimer = xTimerCreateStatic("T1", // Text name for the task.
    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.
}(continues on next page)```
The scheduler has not started yet so a block time is not used.

\[
xReturned = xTimerStart(xTimer, 0);
\]

...  
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(;;);
}

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. The timer period must be greater than 0.

- `xAutoReload` – If `xAutoReload` is set to `pdTRUE` then the timer will expire repeatedly with a frequency set by the `xTimerPeriodInTicks` parameter. If `xAutoReload` 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.

Returns

If the timer is created then a handle to the created timer is returned. If `pxTimerBuffer` was NULL then NULL is returned.

```c
void pvTimerGetTimerID(const TimerHandle_t xTimer)
```

Returns the ID assigned to the timer.

IDs are assigned to timers using the `pvTimerID` parameter of the call to `xTimerCreate()` that was used to create the timer, and by calling the `vTimerSetTimerID()` API function.

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:

See the `xTimerCreate()` API function example usage scenario.

Parameters `xTimer` – The timer being queried.

Returns The ID assigned to the timer being queried.
void **TimerSetTimerID**(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:

See the xTimerCreate() API function example usage scenario.

**Parameters**

- **xTimer** - The timer being updated.
- **pvNewID** - The ID to assign to the timer.

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:

```c
/* // 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.
    }
}
```

**Parameters**  

- **xTimer** - The timer being queried.

**Returns**  

pdFALSE will be returned if the timer is dormant. A value other than pdFALSE will be returned if the timer is active.

TaskHandle_t **xTimerGetTimerDaemonTaskHandle**(void)

Simply returns the handle of the timer service/daemon task. It it not valid to call xTimerGetTimerDaemonTaskHandle() before the scheduler has been started.

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 pended 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:

```c
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...
}

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 );
}
```

**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.

**Returns**
pdPASS is returned if the message was successfully sent to the timer daemon task, other-
wise pdFALSE is returned.

```c
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’).

**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 re-
main 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.

**Returns**
pdPASS is returned if the message was successfully sent to the timer daemon task, other-
wise pdFALSE is returned.

```c
const char* pcTimerGetName (TimerHandle_t xTimer)
```

Returns the name that was assigned to a timer when the timer was created.

**Parameters**  
**xTimer** — The handle of the timer being queried.

**Returns**
The name assigned to the timer specified by the xTimer parameter.

```c
void vTimerSetReloadMode (TimerHandle_t xTimer, const BaseType_t xAutoReload)
```

Updates a timer to be either an auto-reload timer, in which case the timer automatically resets itself each time
it expires, or a one-shot timer, in which case the timer will only expire once unless it is manually restarted.

**Parameters**

- **xTimer** — The handle of the timer being updated.
- **xAutoReload** — If xAutoReload is set to pdTRUE then the timer will expire repeatedly
  with a frequency set by the timer’s period (see the xTimerPeriodInTicks parameter of
  the xTimerCreate() API function). If xAutoReload is set to pdFALSE then the timer will
  be a one-shot timer and enter the dormant state after it expires.

```c
BaseType_t xTimerGetReloadMode (TimerHandle_t xTimer)
```

Queries a timer to determine if it is an auto-reload timer, in which case the timer automatically resets itself
each time it expires, or a one-shot timer, in which case the timer will only expire once unless it is manually restarted.

**Parameters**  
**xTimer** — The handle of the timer being queried.

**Returns**
If the timer is an auto-reload timer then pdTRUE is returned, otherwise pdFALSE is
returned.

```c
UBaseType_t uxTimerGetReloadMode (TimerHandle_t xTimer)
```

Queries a timer to determine if it is an auto-reload timer, in which case the timer automatically resets itself
each time it expires, or a one-shot timer, in which case the timer will only expire once unless it is manually restarted.

**Parameters**  
**xTimer** — The handle of the timer being queried.

**Returns**
If the timer is an auto-reload timer then pdTRUE is returned, otherwise pdFALSE is
returned.

```c
TickType_t xTimerGetPeriod (TimerHandle_t xTimer)
```

Returns the period of a timer.
Chapter 2. API Reference

Parameters `xTimer` - The handle of the timer being queried.

Returns The period of the timer in ticks.

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.

Parameters `xTimer` - The handle of the timer being queried.

Returns 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.

BaseType_t `xTimerGetStaticBuffer` (TimerHandle_t `xTimer`, StaticTimer_t **`ppxTimerBuffer`)

Retrieve pointer to a statically created timer’s data structure buffer. This is the same buffer that is supplied at the time of creation.

Parameters
  • `xTimer` - The timer for which to retrieve the buffer.
  • `ppxTimerBuffer` - Used to return a pointer to the timer’s data structure buffer.

Returns `pdTRUE` if the buffer was retrieved, `pdFALSE` otherwise.

void `vApplicationGetTimerTaskMemory` (StaticTask_t**`ppxTimerTaskTCBBuffer`, StackType_t**`ppxTimerTaskStackBuffer`, uint32_t*`pulTimerTaskStackSize`)

This function is used to provide a statically allocated block of memory to FreeRTOS to hold the Timer Task TCB. This function is required when `configSUPPORT_STATIC_ALLOCATION` is set. For more information see this URI: https://www.FreeRTOS.org/a00110.html#configSUPPORT_STATIC_ALLOCATION

Parameters
  • `ppxTimerTaskTCBBuffer` - A handle to a statically allocated TCB buffer
  • `ppxTimerTaskStackBuffer` - A handle to a statically allocated Stack buffer for the idle task
  • `pulTimerTaskStackSize` - A pointer to the number of elements that will fit in the allocated stack buffer

Macros

`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:

See the `xTimerCreate()` API function example usage scenario.

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.

**Returns** 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.

**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:

See the xTimerCreate() API function example usage scenario.

**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.

**Returns** 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.

**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
(continues on next page)
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.
        }
    }
}

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.

Returns 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.

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:

See the xTimerChangePeriod() API function example usage scenario.

**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.

**Returns** 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.

**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:

```c
/* // 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 );
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", ( 5000 / portTICK_PERIOD_MS), pdFALSE, 0, vBacklightTimerCallback );

    if( xBacklightTimer == NULL ) {
        // The timer was not created.
    } else {
        // Start the timer. No block time is specified, and even if one was specified, 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.
    vTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}

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.

**Returns** `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.

**xTimerStartFromISR (xTimer, pxHigherPriorityTaskWoken)**

A version of `xTimerStart()` that can be called from an interrupt service routine.

**Example usage:**

```c
* // 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
```

(continues on next page)
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.

**Returns**

`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.

### xTimerStopFromISR(xTimer, pxHigherPriorityTaskWoken)

A version of `xTimerStop()` that can be called from an interrupt service routine.

Example usage:

```c
/* // 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. 
*/
```

(continues on next page)
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 call-
ing 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 cur-
tently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken
will get set to pdTRUE internally within the xTimerStopFromISR() function. If xTimer-
StopFromISR() sets this value to pdTRUE then a context switch should be performed
before the interrupt exits.

**Returns** 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 ser-
vice/daemon task relative to other tasks in the system. The timer service/daemon task priority
is set by the configTIMER_TASK_PRIORITY configuration constant.

**xTimerChangePeriodFromISR** (xTimer, xNewPeriod, pxHigherPriorityTaskWoken)
A version of xTimerChangePeriod() that can be called from an interrupt service routine.

**Example usage:**

```c
if( xHigherPriorityTaskWoken != pdFALSE )
{
    // Call the interrupt safe yield function here (actual function
    // depends on the FreeRTOS port being used).
}
```

(continues on next page)
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 \( \frac{500}{\text{portTICK_PERIOD_MS}} \) 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.

**Returns**

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.

**xTimerResetFromISR** (xTimer, pxHigherPriorityTaskWoken)

A version of xTimerReset() that can be called from an interrupt service routine.

Example usage:

```c
// 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.

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;
    // Call the interrupt safe yield function here (actual function
    // depends on the FreeRTOS port being used).
}
```

(continues on next page)
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".

```c
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).*

```c
if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {
    // The reset command was not executed successfully. Take appropriate action here.
}
```

*xHigherPriorityTaskWoken is a pointer to a Boolean variable that is used to track if an interrupt service routine was interrupted by a context switch.*

```c
// Perform the rest of the key processing here.
```

```c
// 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.
```

```c
if( xHigherPriorityTaskWoken != pdFALSE ) {
    // Call the interrupt safe yield function here (actual function depends on the FreeRTOS port being used).
}
```

```c
_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.*

Returns 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.

Type Definitions

typedef struct tmrTimerControl *TimerHandle_t

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

- components/freertos/FreeRTOS-Kernel/include/freertos/event_groups.h
- This header file can be included with:

```c
#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 group is created using xEventGroupCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see https://www.FreeRTOS.org/a00111.html). If an event group is created using xEventGroupCreateStatic() 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:

```c
// 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?
if( xCreatedEventGroup == NULL )
{
    // The event group was not created because there was insufficient
    // FreeRTOS heap available.
} else
{
    // The event group was created.
}
```

**Returns** 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 https://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 xEventGroupCreate() then the
required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see
https://www.FreeRTOS.org/a00111.html). If an event group is created using xEventGroupCreateStatic() then
the application writer must instead provide the memory that will get used by the event group. xEventGroupCre-
ateStatic() 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 avail-
able 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 confi-
gUSE_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:

```c
// 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);
```

**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.

**Returns** If the event group was created then a handle to the event group is returned. If pxEvent-
GroupBuffer was NULL then NULL is returned.

```c
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:

```c
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

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...
        pdTRUE,      // BIT_0 and BIT_4 should be cleared before...
    );
```

(continues on next page)
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.
  }
}

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. A value of portMAX_DELAY can be used to block indefinitely (provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h).

Returns

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.

```
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.
    }
}

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.

Returns The value of the event group before the specified bits were cleared.

EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet )

Set bits within an event group. This function cannot be called from an interrupt. xEventGroupSetBits-
FromISR() 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:

```c
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    // 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.
}

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.

Returns 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.

**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:

```c
// Bits used by the three tasks.
#define TASK_0_BIT   ( 1 << 0 )
#define TASK_1_BIT   ( 1 << 1 )
#define TASK_2_BIT   ( 1 << 2 )
```

(continues on next page)
#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 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.

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.

Returns
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.

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:

```c
#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 )
{
    // 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.
        portYIELD_FROM_ISR(pdTRUE);
    }
}
```

**Note:** If this function returns pdPASS then the timer task is ready to run and a
`portYIELD_FROM_ISR(pdTRUE)` should be executed to perform the needed clear on the event group. This
behavior is different from `xEventGroupSetBitsFromISR` because the parameter `xHigherPriorityTaskWoken`
is not present.

### 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.

### Returns
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.

### 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 `xEventGroupSetBitsFromISR()` 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:

```c
#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;
    (continues on next page)```
// 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 );
}

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.

Returns 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.

**xEventGroupGetBits** (xEventGroup)

Returns the current value of the bits in an event group. This function cannot be used from an interrupt.

Parameters

• xEventGroup – The event group being queried.

Returns The event group bits at the time xEventGroupGetBits() was called.

Type Definitions
typedef struct EventGroupDef_t *EventGroupHandle_t

typedef TickType_t EventBits_t

Stream Buffer API

Header File

• components/freertos/FreeRTOS-Kernel/include/freertos/stream_buffer.h
• This header file can be included with:
#include "freertos/stream_buffer.h"

## Functions

### BaseType_t xStreamBufferGetStaticBuffers

```c
StreamBufferHandle_t xStreamBuffer, uint8_t **ppucStreamBufferStorageArea, StaticStreamBuffer_t **ppxStaticStreamBuffer)
```

Retrieve pointers to a statically created stream buffer’s data structure buffer and storage area buffer. These are the same buffers that are supplied at the time of creation.

#### Parameters

- **xStreamBuffer**: The stream buffer for which to retrieve the buffers.
- **ppucStreamBufferStorageArea**: Used to return a pointer to the stream buffer’s storage area buffer.
- **ppxStaticStreamBuffer**: Used to return a pointer to the stream buffer’s data structure buffer.

#### Returns

- **pdTRUE** if buffers were retrieved, **pdFALSE** otherwise.

### size_t xStreamBufferSend

```c
StreamBufferHandle_t xStreamBuffer, const void *pvTxData, size_t xDataLengthBytes, TickType_t xTicksToWait)
```

Sends bytes to a stream buffer. The bytes are copied into the stream buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as `xStreamBufferSend()`) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as `xStreamBufferReceive()`) inside a critical section and set the receive block time to 0.

Use `xStreamBufferSend()` to write to a stream buffer from a task. Use `xStreamBufferSendFromISR()` to write to a stream buffer from an interrupt service routine (ISR).

#### Example use:

```c
void vAFunction( StreamBufferHandle_t xStreamBuffer )
{
    size_t xBytesSent;
    uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
    char *pcStringToSend = "String to send";
    const TickType_t x100ms = pdMS_TO_TICKS( 100 );

    // Send an array to the stream buffer, blocking for a maximum of 100ms to
    // wait for enough space to be available in the stream buffer.
    xBytesSent = xStreamBufferSend( xStreamBuffer, ( void * ) ucArrayToSend,...
                                 sizeof( ucArrayToSend ), x100ms );

    if( xBytesSent != sizeof( ucArrayToSend ) )
    {
        // The call to xStreamBufferSend() times out before there was enough
        // space in the buffer for the data to be written, but it did
        // successfully write xBytesSent bytes.
    }

    // Send the string to the stream buffer. Return immediately if there is not
    // enough space in the buffer.
    xBytesSent = xStreamBufferSend( xStreamBuffer, ( void * ) pcStringToSend,...
                                 strlen( pcStringToSend ), 0 );

    // (continues on next page)
```
if( xBytesSent != strlen( pcStringToSend ) )
{
    // The entire string could not be added to the stream buffer because
    // there was not enough free space in the buffer, but xBytesSent bytes
    // were sent. Could try again to send the remaining bytes.
}

Parameters

• **xStreamBuffer** - The handle of the stream buffer to which a stream is being sent.
• **pvTxData** - A pointer to the buffer that holds the bytes to be copied into the stream buffer.
• **xDataLengthBytes** - The maximum number of bytes to copy from pvTxData into the stream buffer.
• **xTicksToWait** - The maximum amount of time the task should remain in the Blocked state to wait for enough space to become available in the stream buffer, should the stream buffer contain too little space to hold the another xDataLengthBytes bytes. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible. A task does not use any CPU time when it is in the blocked state.

**Returns** The number of bytes written to the stream buffer. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible.

```c
size_t xStreamBufferSendFromISR (StreamBufferHandle_t xStreamBuffer, const void *pvTxData, size_t xDataLengthBytes, BaseType_t *const pxHigherPriorityTaskWoken)
```

Interrupt safe version of the API function that sends a stream of bytes to the stream buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write to a stream buffer from an interrupt service routine (ISR).

Example use:

```c
// A stream buffer that has already been created.
StreamBufferHandle_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
    size_t xBytesSent;
    char *pcStringToSend = "String to send";
    BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

    // Attempt to send the string to the stream buffer.
```
xBytesSent = xStreamBufferSendFromISR( xStreamBuffer,  
    (void *) pcStringToSend,  
    strlen( pcStringToSend ),  
    &xHigherPriorityTaskWoken );

if( xBytesSent != strlen( pcStringToSend ) )
{
    // There was not enough free space in the stream buffer for the entire  
    // string to be written, ut xBytesSent bytes were written.
}

// If xHigherPriorityTaskWoken was set to pdTRUE inside  
// xStreamBufferSendFromISR() then a task that has a priority above the  
// priority of the currently executing task was unblocked and a context  
// switch should be performed to ensure the ISR returns to the unblocked  
// task. In most FreeRTOS ports this is done by simply passing  
// xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the  
// variables value, and perform the context switch if necessary. Check the  
// documentation for the port in use for port specific instructions.
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

Parameters
• xStreamBuffer – The handle of the stream buffer to which a stream is being sent.
• pvTxData – A pointer to the data that is to be copied into the stream buffer.
• xDataLengthBytes – The maximum number of bytes to copy from pvTxData into the  
    stream buffer.
• pxHigherPriorityTaskWoken – It is possible that a stream buffer will have a task  
    blocked on it waiting for data. Calling xStreamBufferSendFromISR() can make data available,  
    and so cause a task that was waiting for data to leave the Blocked state. If calling  
    xStreamBufferSendFromISR() causes a task to leave the Blocked state, and the unblocked  
    task has a priority higher than the currently executing task (the task that was interrupted),  
    then, internally, xStreamBufferSendFromISR() will set *pxHigherPriorityTaskWoken to  
    pdTRUE. If xStreamBufferSendFromISR() sets this value to pdTRUE, then normally a context switch  
    should be performed before the interrupt is exited. This will ensure that the interrupt returns  
    directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to  
    pdFALSE before it is passed into the function. See the example code below for an example.

Returns The number of bytes actually written to the stream buffer, which will be less than xDataLengthBytes  
    if the stream buffer didn’t have enough free space for all the bytes to be written.

size_t xStreamBufferReceive (StreamBufferHandle_t xStreamBuffer, void *pvRxData, size_t  
    xBufferLengthBytes, TickType_t xTicksToWait)

Receives bytes from a stream buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation,  
    as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that  
    will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It  
    is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not  
    safe to have multiple different writers or multiple different readers. If there are to be multiple different writers  
    then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside  
    a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then  
    the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside  
    a critical section and set the receive block time to 0.

Use xStreamBufferReceive() to read from a stream buffer from a task. Use xStreamBufferReceiveFromISR()  
    to read from a stream buffer from an interrupt service routine (ISR).
Example use:

```c
void vAFunction( StreamBuffer_t xStreamBuffer )
{
    uint8_t ucRxData[ 20 ];
    size_t xReceivedBytes;
    const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

    // Receive up to another sizeof( ucRxData ) bytes from the stream buffer.
    // Wait in the Blocked state (so not using any CPU processing time) for a
    // maximum of 100ms for the full sizeof( ucRxData ) number of bytes to be
    // available.
    xReceivedBytes = xStreamBufferReceive( xStreamBuffer,
                                          ( void * ) ucRxData,
                                          sizeof( ucRxData ),
                                          xBlockTime );

    if( xReceivedBytes > 0 )
    {
        // A ucRxData contains another xReceivedBytes bytes of data, which can
        // be processed here....
    }
}
```

**Parameters**

- **xStreamBuffer** - The handle of the stream buffer from which bytes are to be received.
- **pvRxData** - A pointer to the buffer into which the received bytes will be copied.
- **xBuffeLengthBytes** - The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes.
- **xBitsToWait** - The maximum amount of time the task should remain in the Blocked state to wait for data to become available if the stream buffer is empty. xStreamBuffer-Receive() will return immediately if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTickToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. A task does not use any CPU time when it is in the Blocked state.

**Returns**

The number of bytes actually read from the stream buffer, which will be less than xBufferLengthBytes if the call to xStreamBufferReceive() timed out before xBufferLengthBytes were available.

```c
size_t xStreamBufferReceiveFromISR( StreamBufferHandle_t xStreamBuffer, void *pvRxData, size_t xBufferLengthBytes, BaseType_t *const pxHigherPriorityTaskWoken )
```

An interrupt safe version of the API function that receives bytes from a stream buffer.

Use xStreamBufferReceive() to read bytes from a stream buffer from a task. Use xStreamBufferReceive-FromISR() to read bytes from a stream buffer from an interrupt service routine (ISR).

Example use:

```c
// A stream buffer that has already been created.
StreamBuffer_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
    uint8_t ucRxData[ 20 ];
    size_t xReceivedBytes;
    (continues on next page)
```
## Chapter 2. API Reference

(continued from previous page)

```c
BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

// Receive the next stream from the stream buffer.
xReceivedBytes = xStreamBufferReceiveFromISR( xStreamBuffer,
    { void * ) ucRxData,
    sizeof( ucRxData ),
    &xHigherPriorityTaskWoken );

if( xReceivedBytes > 0 )
{
    // ucRxData contains xReceivedBytes read from the stream buffer.
    // Process the stream here....
}

// If xHigherPriorityTaskWoken was set to pdTRUE inside
// xStreamBufferReceiveFromISR() then a task that has a priority above the
// priority of the currently executing task was unblocked and a context
// switch should be performed to ensure the ISR returns to the unblocked
// task. In most FreeRTOS ports this is done by simply passing
// xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
// variables value, and perform the context switch if necessary. Check the
// documentation for the port in use for port specific instructions.
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

### Parameters

- **xStreamBuffer** – The handle of the stream buffer from which a stream is being received.
- **pvRxData** – A pointer to the buffer into which the received bytes are copied.
- **xBuffLengthBytes** – The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes.
- **pxHigherPriorityTaskWoken** – It is possible that a stream buffer will have a task blocked on it waiting for space to become available. Calling xStreamBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xStreamBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferReceiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferReceiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

### Returns

The number of bytes read from the stream buffer, if any.

```c
void vStreamBufferDelete (StreamBufferHandle_t xStreamBuffer)

Deletes a stream buffer that was previously created using a call to xStreamBufferCreate() or xStreamBufferCreateStatic(). If the stream buffer was created using dynamic memory (that is, by xStreamBufferCreate()), then the allocated memory is freed.

A stream buffer handle must not be used after the stream buffer has been deleted.
```

### Parameters **xStreamBuffer** – The handle of the stream buffer to be deleted.

```c
BaseType_t xStreamBufferIsFull (StreamBufferHandle_t xStreamBuffer)

Queries a stream buffer to see if it is full. A stream buffer is full if it does not have any free space, and therefore cannot accept any more data.
```

### Parameters **xStreamBuffer** – The handle of the stream buffer being queried.

### Returns

If the stream buffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.
BaseType_t *xStreamBufferIsEmpty (StreamBufferHandle_t xStreamBuffer)

Queries a stream buffer to see if it is empty. A stream buffer is empty if it does not contain any data.

Parameters xStreamBuffer – The handle of the stream buffer being queried.

Returns If the stream buffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

BaseType_t *xStreamBufferReset (StreamBufferHandle_t xStreamBuffer)

Resets a stream buffer to its initial, empty, state. Any data that was in the stream buffer is discarded. A stream buffer can only be reset if there are no tasks blocked waiting to either send to or receive from the stream buffer.

Parameters xStreamBuffer – The handle of the stream buffer being queried.

Returns If the stream buffer is reset then pdPASS is returned. If there was a task blocked waiting to send to or read from the stream buffer then the stream buffer is not reset and pdFAIL is returned.

size_t *xStreamBufferSpacesAvailable (StreamBufferHandle_t xStreamBuffer)

Queries a stream buffer to see how much free space it contains, which is equal to the amount of data that can be sent to the stream buffer before it is full.

Parameters xStreamBuffer – The handle of the stream buffer being queried.

Returns The number of bytes that can be written to the stream buffer before the stream buffer would be full.

size_t *xStreamBufferBytesAvailable (StreamBufferHandle_t xStreamBuffer)

Queries a stream buffer to see how much data it contains, which is equal to the number of bytes that can be read from the stream buffer before the stream buffer would be empty.

Parameters xStreamBuffer – The handle of the stream buffer being queried.

Returns The number of bytes that can be read from the stream buffer before the stream buffer would be empty.

BaseType_t *xStreamBufferSetTriggerLevel (StreamBufferHandle_t xStreamBuffer, size_t xTriggerLevel)

A stream buffer’s trigger level is the number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task’s block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task’s block time expires. If a reading task’s block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.

A trigger level is set when the stream buffer is created, and can be modified using xStreamBufferSetTriggerLevel().

Parameters

• xStreamBuffer – The handle of the stream buffer being updated.
• xTriggerLevel – The new trigger level for the stream buffer.

Returns If xTriggerLevel was less than or equal to the stream buffer’s length then the trigger level will be updated and pdTRUE is returned. Otherwise pdFALSE is returned.

BaseType_t *xStreamBufferSendCompletedFromISR (StreamBufferHandle_t xStreamBuffer, BaseType_t *pxHigherPriorityTaskWoken)

For advanced users only.

The sbSEND_COMPLETED() macro is called from within the FreeRTOS APIs when data is sent to a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbSEND_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xStreamBufferSendCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbSEND_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.
See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**
- **xStreamBuffer** - The handle of the stream buffer to which data was written.
- **pxHigherPriorityTaskWoken** - *pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xStreamBufferSendCompletedFromISR(). If calling xStreamBufferSendCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then *pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR.

**Returns** If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

```c
BaseType_t xStreamBufferSendCompletedFromISR( StreamBufferHandle_t xStreamBuffer,
                                          BaseType_t *pxHigherPriorityTaskWoken)
```

For advanced users only.

The sbRECEIVE_COMPLETED() macro is called from within the FreeRTOS APIs when data is read out of a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbRECEIVE_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xStreamBufferReceiveCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbRECEIVE_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**
- **xStreamBuffer** - The handle of the stream buffer from which data was read.
- **pxHigherPriorityTaskWoken** - *pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xStreamBufferReceiveCompletedFromISR(). If calling xStreamBufferReceiveCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then *pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR.

**Returns** If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

```c
Macros
```
```
xStreamBufferCreateWithCallback (xBufferSizeBytes, xTriggerLevelBytes,
                      pxSendCompletedCallback, pxReceiveCompletedCallback)
```

Creates a new stream buffer using dynamically allocated memory. See xStreamBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xStreamBufferCreate() to be available.

Example use:

```c
void vAFunction( void )
{
    StreamBufferHandle_t xStreamBuffer;
    const size_t xStreamBufferSizeBytes = 100, xTriggerLevel = 10;

    // Create a stream buffer that can hold 100 bytes. The memory used to hold
    // both the stream buffer structure and the data in the stream buffer is
    // allocated dynamically.
    xStreamBuffer = xStreamBufferCreate( xStreamBufferSizeBytes, xTriggerLevel );

    if( xStreamBuffer == NULL )
```
{  // There was not enough heap memory space available to create the  // stream buffer.
}
else{
  // The stream buffer was created successfully and can now be used.
}

Parameters

- **xBuffersizeBytes** – The total number of bytes the stream buffer will be able to hold at any one time.
- **xTriggerLevelBytes** – The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task’s block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task’s block time expires. If a reading task’s block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.
- **pxSendCompletedCallback** – Callback invoked when number of bytes at least equal to trigger level is sent to the stream buffer. If the parameter is NULL, it will use the default implementation provided by sbSEND_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.
- **pxReceiveCompletedCallback** – Callback invoked when more than zero bytes are read from a stream buffer. If the parameter is NULL, it will use the default implementation provided by sbRECEIVE_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.

Returns

If NULL is returned, then the stream buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the stream buffer data structures and storage area. A non-NULL value being returned indicates that the stream buffer has been created successfully - the returned value should be stored as the handle to the created stream buffer.

**xStreamBufferCreateStaticWithCallback** (xBuffersizeBytes, xTriggerLevelBytes, pucStreamBufferStorageArea, pxStaticStreamBuffer, pxSendCompletedCallback, pxReceiveCompletedCallback)

Creates a new stream buffer using statically allocated memory. See **xStreamBufferCreate()** for a version that uses dynamically allocated memory.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for **xStreamBufferCreateStatic()** to be available.

Example use:

```c
// Used to dimension the array used to hold the streams. The available space  // will actually be one less than this, so 999.
#define STORAGE_SIZE_BYTES 1000

// Defines the memory that will actually hold the streams within the stream  // buffer.
static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];
```

(continues on next page)
// The variable used to hold the stream buffer structure.
StaticStreamBuffer_t xStreamBufferStruct;

void MyFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xTriggerLevel = 1;

xStreamBuffer = xStreamBufferCreateStatic( sizeof( ucStorageBuffer ),
                                          xTriggerLevel,
                                          ucStorageBuffer,
                                          &xStreamBufferStruct );

// As neither the pucStreamBufferStorageArea or pxStaticStreamBuffer
// parameters were NULL, xStreamBuffer will not be NULL, and can be used to
// reference the created stream buffer in other stream buffer API calls.

// Other code that uses the stream buffer can go here.
}

Parameters

- **xBufferSizeBytes** – The size, in bytes, of the buffer pointed to by the pucStreamBufferStorageArea parameter.
- **xTriggerLevelBytes** – The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer waits for new data to be moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task’s block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task’s block time expires. If a reading task’s block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.
- **pucStreamBufferStorageArea** – Must point to a uint8_t array that is at least xBufferSizeBytes big. This is the array to which streams are copied when they are written to the stream buffer.
- **pxStaticStreamBuffer** – Must point to a variable of type StaticStreamBuffer_t, which will be used to hold the stream buffer’s data structure.
- **pxSendCompletedCallback** – Callback invoked when number of bytes at least equal to trigger level is sent to the stream buffer. If the parameter is NULL, it will use the default implementation provided by sbSEND_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.
- **pxReceiveCompletedCallback** – Callback invoked when more than zero bytes are read from a stream buffer. If the parameter is NULL, it will use the default implementation provided by sbRECEIVE_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.

Returns

If the stream buffer is created successfully then a handle to the created stream buffer is returned. If either pucStreamBufferStorageArea or pxStaticStreamBuffer are NULL then NULL is returned.

Type Definitions

typedef struct StreamBufferDef_t *StreamBufferHandle_t
typedef void (*StreamBufferCallbackFunction_t)(StreamBufferHandle_t xStreamBuffer, BaseType_t xIsInsideISR, BaseType_t *const pxHigherPriorityTaskWoken)

Type used as a stream buffer’s optional callback.

Message Buffer API

Header File

- components/freertos/FreeRTOS-Kernel/include/freertos/message_buffer.h
- This header file can be included with:

```
#include "freertos/message_buffer.h"
```

Macros

`xMessageBufferCreateWithCallback` (xBufferSizeBytes, pxSendCompletedCallback, pxReceiveCompletedCallback)

Creates a new message buffer using dynamically allocated memory. See `xMessageBufferCreateStatic()` for a version that uses statically allocated memory (memory that is allocated at compile time).

`configSUPPORT_DYNAMIC_ALLOCATION` must be set to 1 or left undefined in `FreeRTOSConfig.h` for `xMessageBufferCreate()` to be available.

Example use:

```c
void vAFunction( void )
{
MessageBufferHandle_t xMessageBuffer;
const size_t xMessageBufferSizeBytes = 100;

// Create a message buffer that can hold 100 bytes. The memory used to hold
// both the message buffer structure and the messages themselves is allocated
// dynamically. Each message added to the buffer consumes an additional 4
// bytes which are used to hold the length of the message.
xMessageBuffer = xMessageBufferCreate( xMessageBufferSizeBytes );

if( xMessageBuffer == NULL )
{
    // There was not enough heap memory space available to create the
    // message buffer.
}
else
{
    // The message buffer was created successfully and can now be used.
}
```

Parameters

- `xBufferSizeBytes` - The total number of bytes (not messages) the message buffer will be able to hold at any one time. When a message is written to the message buffer an additional `sizeof( size_t )` bytes are also written to store the message’s length. `sizeof( size_t )` is typically 4 bytes on a 32-bit architecture, so on most 32-bit architectures a 10 byte message will take up 14 bytes of message buffer space.
- `pxSendCompletedCallback` - Callback invoked when a send operation to the message buffer is complete. If the parameter is NULL or `xMessageBufferCreate()` is called without the parameter, then it will use the default implementation provided by `sbSEND_COMPLETED` macro. To enable the callback, `configUSE_SB_COMPLETED_CALLBACK` must be set to 1 in `FreeRTOSConfig.h`. 
**pxReceiveCompletedCallback** — Callback invoked when a receive operation from the message buffer is complete. If the parameter is NULL or xMessageBufferCreate() is called without the parameter, it will use the default implementation provided by sbRECEIVE_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.

**Returns** If NULL is returned, then the message buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the message buffer data structures and storage area. A non-NULL value being returned indicates that the message buffer has been created successfully - the returned value should be stored as the handle to the created message buffer.

**xMessageBufferCreateStaticWithCallback**(xBuffSizeBytes, pucMessageBufferStorageArea, pxStaticMessageBuffer, pxSendCompletedCallback, pxReceiveCompletedCallback)

Creates a new message buffer using statically allocated memory. See xMessageBufferCreate() for a version that uses dynamically allocated memory.

Example use:

```c
// Used to dimension the array used to hold the messages. The available space // will actually be one less than this, so 999.
#define STORAGE_SIZE_BYTES 1000

// Define the memory that will actually hold the messages within the message // buffer.
static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];

// The variable used to hold the message buffer structure.
StaticMessageBuffer_t xMessageBufferStruct;

void MyFunction( void )
{
    MessageBufferHandle_t xMessageBuffer;

    xMessageBuffer = xMessageBufferCreateStatic( sizeof( ucStorageBuffer ),
                                              ucStorageBuffer,
                                              &xMessageBufferStruct );

    // As neither the pucMessageBufferStorageArea or pxStaticMessageBuffer // parameters were NULL, xMessageBuffer will not be NULL, and can be used to // reference the created message buffer in other message buffer API calls.

    // Other code that uses the message buffer can go here.
}
```

**Parameters**

- **xBuffSizeBytes** — The size, in bytes, of the buffer pointed to by the pucMessageBufferStorageArea parameter. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message’s length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture a 10 byte message will take up 14 bytes of message buffer space. The maximum number of bytes that can be stored in the message buffer is actually (xBuffSizeBytes - 1).

- **pucMessageBufferStorageArea** — Must point to a uint8_t array that is at least xBuffSizeBytes big. This is the array to which messages are copied when they are written to the message buffer.

- **pxStaticMessageBuffer** — Must point to a variable of type StaticMessageBuffer_t, which will be used to hold the message buffer’s data structure.

- **pxSendCompletedCallback** — Callback invoked when a new message is sent to the message buffer. If the parameter is NULL or xMessageBufferCre-
ate() is called without the parameter, then it will use the default implementation provided by sbSEND_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.

- **pxReceiveCompletedCallback** — Callback invoked when a message is read from a message buffer. If the parameter is NULL or xMessageBufferCreate() is called without the parameter, it will use the default implementation provided by sbRECEIVE_COMPLETED macro. To enable the callback, configUSE_SB_COMPLETED_CALLBACK must be set to 1 in FreeRTOSConfig.h.

**Returns** If the message buffer is created successfully then a handle to the created message buffer is returned. If either pucMessageBufferStorageArea or pxStaticmessageBuffer are NULL then NULL is returned.

**xMessageBufferGetStaticBuffers** (xMessageBuffer, ppucMessageBufferStorageArea, ppxStaticMessageBuffer)

Retrieve pointers to a statically created message buffer’s data structure buffer and storage area buffer. These are the same buffers that are supplied at the time of creation.

**Parameters**
- **xMessageBuffer** — The message buffer for which to retrieve the buffers.
- **ppucMessageBufferStorageArea** — Used to return a pointer to the message buffer’s storage area buffer.
- **ppxStaticMessageBuffer** — Used to return a pointer to the message buffer’s data structure buffer.

**Returns** pdTRUE if buffers were retrieved, pdFALSE otherwise.

**xMessageBufferSend** (xMessageBuffer, pvTxData, xDataLengthBytes, xTicksToWait)

Sends a discrete message to the message buffer. The message can be any length that fits within the buffer’s free space, and is copied into the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

**Example use:**

```c
void vAFunction( MessageBufferHandle_t xMessageBuffer )
{
    size_t xBytesSent;
    uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
    char *pcStringToSend = "String to send";
    const TickType_t x100ms = pdMS_TO_TICKS( 100 );

    // Send an array to the message buffer, blocking for a maximum of 100ms to
    // wait for enough space to be available in the message buffer.
    xBytesSent = xMessageBufferSend( xMessageBuffer, ( void * ) ucArrayToSend, _
                        sizeof( ucArrayToSend ), x100ms );
    if( xBytesSent != sizeof( ucArrayToSend ) )
    {
        // The call to xMessageBufferSend() times out before there was enough
```
xBytesSent = xMessageBufferSend(xMessageBuffer, (void *) pcStringToSend, strlen(pcStringToSend), 0);

if (xBytesSent != strlen(pcStringToSend)) {
    // The string could not be added to the message buffer because there was not enough free space in the buffer.
}

Parameters

- **xMessageBuffer** - The handle of the message buffer to which a message is being sent.
- **pvTxData** - A pointer to the message that is to be copied into the message buffer.
- **xDataLengthBytes** - The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message’s length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length).
- **xTicksToWait** - The maximum amount of time the calling task should remain in the Blocked state to wait for enough space to become available in the message buffer, should the message buffer have insufficient space when xMessageBufferSend() is called. The calling task will never block if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state.

Returns

The number of bytes written to the message buffer. If the call to xMessageBufferSend() times out before there was enough space to write the message into the message buffer then zero is returned. If the call did not time out then xDataLengthBytes is returned.

xMessageBufferSendFromISR(xMessageBuffer, pvTxData, xDataLengthBytes, pxHigherPriorityTaskWoken)

Interrupt safe version of the API function that sends a discrete message to the message buffer. The message can be any length that fits within the buffer’s free space, and is copied into the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

Example use:
// A message buffer that has already been created.
MessageBufferHandle_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
    size_t xBytesSent;
    char *pcStringToSend = "String to send";
    BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

    // Attempt to send the string to the message buffer.
    xBytesSent = xMessageBufferSendFromISR( xMessageBuffer,
                                             ( void * ) pcStringToSend,
                                             strlen( pcStringToSend ),
                                             &xHigherPriorityTaskWoken );

    if( xBytesSent != strlen( pcStringToSend ) )
    {
        // The string could not be added to the message buffer because there was
        // not enough free space in the buffer.
    }

    // If xHigherPriorityTaskWoken was set to pdTRUE inside
    // xMessageBufferSendFromISR() then a task that has a priority above the
    // priority of the currently executing task was unblocked and a context
    // switch should be performed to ensure the ISR returns to the unblocked
    // task. In most FreeRTOS ports this is done by simply passing
    // xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
    // variables value, and perform the context switch if necessary. Check the
    // documentation for the port in use for port specific instructions.
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

Parameters

- **xMessageBuffer** – The handle of the message buffer to which a message is being sent.
- **pvTxData** – A pointer to the message that is to be copied into the message buffer.
- **xDataLengthBytes** – The length of the message. That is, the number of bytes to copy
  from pvTxData into the message buffer. When a message is written to the message buffer
  an additional sizeof(size_t) bytes are also written to store the message’s length. sizeof(size_t)
  is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting
  xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20
  bytes of message data and 4 bytes to hold the message length).
- **pxHigherPriorityTaskWoken** – It is possible that a message buffer will have a
task blocked on it waiting for data. Calling xMessageBufferSendFromISR() can make
data available, and so cause a task that was waiting for data to leave the Blocked state. If
calling xMessageBufferSendFromISR() causes a task to leave the Blocked state, and the
unblocked task has a priority higher than the currently executing task (the task that was
interrupted), then, internally, xMessageBufferSendFromISR() will set *pxHigherPriority-
TaskWoken to pdTRUE. If xMessageBufferSendFromISR() sets this value to pdTRUE,
then normally a context switch should be performed before the interrupt is exited. This
will ensure that the interrupt returns directly to the highest priority Ready state task. *px-
HigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function.
See the code example below for an example.

Returns  The number of bytes actually written to the message buffer. If the message buffer didn’t
have enough free space for the message to be stored then 0 is returned, otherwise xDataL-
enthBytes is returned.

**xMessageBufferReceive** (xMessageBuffer, pvRxData, xBufferLengthBytes, xTicksToWait)

Receives a discrete message from a message buffer. Messages can be of variable length and are copied out of
the buffer.
 chapter 2. API Reference

: uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

Example use:

```c
void vAFunction( MessageBuffer_t xMessageBuffer )
{
    uint8_t ucRxData[ 20 ];
    size_t xReceivedBytes;
    const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

    // Receive the next message from the message buffer. Wait in the Blocked
    // state (so not using any CPU processing time) for a maximum of 100ms for
    // a message to become available.
    xReceivedBytes = xMessageBufferReceive( xMessageBuffer,
        ( void * ) ucRxData,
        sizeof( ucRxData ),
        xBlockTime );

    if( xReceivedBytes > 0 )
    {
        // A ucRxData contains a message that is xReceivedBytes long. Process
        // the message here....
    }
}
```

Parameters

- **xMessageBuffer** – The handle of the message buffer from which a message is being received.
- **pvRxData** – A pointer to the buffer into which the received message is to be copied.
- **xBufferLengthBytes** – The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned.
- **xTicksToWait** – The maximum amount of time the task should remain in the Blocked state to wait for a message, should the message buffer be empty. xMessageBufferReceive() will return immediately if xTicksToWait is zero and the message buffer is empty. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state.

Returns

The length, in bytes, of the message read from the message buffer, if any. If xMessageBufferReceive() times out before a message became available then zero is returned. If the length of the message is greater than xBufferLengthBytes then the message will be left in the message buffer and zero is returned.
An interrupt safe version of the API function that receives a discrete message from a message buffer. Messages can be of variable length and are copied out of the buffer.

: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

Example use:

```c
// A message buffer that has already been created.
MessageBuffer_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
    uint8_t ucRxData[20];
    size_t xReceivedBytes;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

    // Receive the next message from the message buffer.
    xReceivedBytes = xMessageBufferReceiveFromISR( xMessageBuffer,
                                                (void *) ucRxData,
                                                sizeof( ucRxData ),
                                                &xHigherPriorityTaskWoken );

    if( xReceivedBytes > 0 )
    {
        // A ucRxData contains a message that is xReceivedBytes long. Process
        // the message here....
    }

    // If xHigherPriorityTaskWoken was set to pdTRUE inside
    // xMessageBufferReceiveFromISR() then a task that has a priority above the
    // priority of the currently executing task was unblocked and a context
    // switch should be performed to ensure the ISR returns to the unblocked
    // task. In most FreeRTOS ports this is done by simply passing
    // xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
    // variables value, and perform the context switch if necessary. Check the
    // documentation for the port in use for port specific instructions.
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

**Parameters**

- `xMessageBuffer` – The handle of the message buffer from which a message is being received.
- `pvRxData` – A pointer to the buffer into which the received message is to be copied.
- `xBuffLengthBytes` – The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBuffLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned.
- `pxHigherPriorityTaskWoken` – It is possible that a message buffer will have a task
blocked on it waiting for space to become available. Calling `xMessageBufferReceiveFromISR()` can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling `xMessageBufferReceiveFromISR()` causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, `xMessageBufferReceiveFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE`. If `xMessageBufferReceiveFromISR()` sets this value to `pdTRUE`, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. `*pxHigherPriorityTaskWoken` should be set to `pdFALSE` before it is passed into the function. See the code example below for an example.

**Returns**  The length, in bytes, of the message read from the message buffer, if any.

**vMessageBufferDelete** (xMessageBuffer)

Deletes a message buffer that was previously created using a call to `xMessageBufferCreate()` or `xMessageBufferCreateStatic()`. If the message buffer was created using dynamic memory (that is, by `xMessageBufferCreate()`), then the allocated memory is freed.

A message buffer handle must not be used after the message buffer has been deleted.

**Parameters**

- `xMessageBuffer` – The handle of the message buffer to be deleted.

**xMessageBufferIsFull** (xMessageBuffer)

Tests to see if a message buffer is full. A message buffer is full if it cannot accept any more messages, of any size, until space is made available by a message being removed from the message buffer.

**Parameters**

- `xMessageBuffer` – The handle of the message buffer being queried.

**Returns**  If the message buffer referenced by `xMessageBuffer` is full then `pdTRUE` is returned. Otherwise `pdFALSE` is returned.

**xMessageBufferIsEmpty** (xMessageBuffer)

Tests to see if a message buffer is empty (does not contain any messages).

**Parameters**

- `xMessageBuffer` – The handle of the message buffer being queried.

**Returns**  If the message buffer referenced by `xMessageBuffer` is empty then `pdTRUE` is returned. Otherwise `pdFALSE` is returned.

**xMessageBufferReset** (xMessageBuffer)

Resets a message buffer to its initial empty state, discarding any message it contained.

A message buffer can only be reset if there are no tasks blocked on it.

**Parameters**

- `xMessageBuffer` – The handle of the message buffer being reset.

**Returns**  If the message buffer was reset then `pdPASS` is returned. If the message buffer could not be reset because either there was a task blocked on the message queue to wait for space to become available, or to wait for a message to be available, then `pdFAIL` is returned.

**xMessageBufferSpaceAvailable** (xMessageBuffer)

```c
size_t xMessageBufferSpaceAvailable( MessageBufferHandle_t xMessageBuffer );
```

Returns the number of bytes of free space in the message buffer.

**Parameters**

- `xMessageBuffer` – The handle of the message buffer being queried.

**Returns**  The number of bytes that can be written to the message buffer before the message buffer would be full. When a message is written to the message buffer an additional `sizeof(size_t)` bytes are also written to store the message’s length. `sizeof(size_t)` is typically 4 bytes on...
a 32-bit architecture, so if `xMessageBufferSpacesAvailable()` returns 10, then the size of the
largest message that can be written to the message buffer is 6 bytes.

**xMessageBufferSpacesAvailable** *(xMessageBuffer)*

**xMessageBufferNextLengthBytes** *(xMessageBuffer)*

Returns the length (in bytes) of the next message in a message buffer. Useful if `xMessageBufferReceive()` returned 0 because the size of the buffer passed into `xMessageBufferReceive()` was too small to hold the next message.

**Parameters**

- `xMessageBuffer` — The handle of the message buffer being queried.

**Returns**

The length (in bytes) of the next message in the message buffer, or 0 if the message buffer is empty.

**xMessageBufferSendCompletedFromISR** *(xMessageBuffer, pxHigherPriorityTaskWoken)*

For advanced users only.

The `sbSEND_COMPLETED()` macro is called from within the FreeRTOS APIs when data is sent to a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the `sbSEND_COMPLETED()` macro sends a notification to the task to remove it from the Blocked state. `xMessageBufferSendCompletedFromISR()` does the same thing. It is provided to enable application writers to implement their own version of `sbSEND_COMPLETED()`, and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**

- `xMessageBuffer` — The handle of the stream buffer to which data was written.
- `pxHigherPriorityTaskWoken` — `pxHigherPriorityTaskWoken` should be initialised to `pdFALSE` before it is passed into `xMessageBufferSendCompletedFromISR()`. If calling `xMessageBufferSendCompletedFromISR()` removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then `pxHigherPriorityTaskWoken` will get set to `pdTRUE` indicating that a context switch should be performed before exiting the ISR.

**Returns**

If a task was removed from the Blocked state then `pdTRUE` is returned. Otherwise `pdFALSE` is returned.

**xMessageBufferReceiveCompletedFromISR** *(xMessageBuffer, pxHigherPriorityTaskWoken)*

For advanced users only.

The `sbRECEIVE_COMPLETED()` macro is called from within the FreeRTOS APIs when data is read out of a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the `sbRECEIVE_COMPLETED()` macro sends a notification to the task to remove it from the Blocked state. `xMessageBufferReceiveCompletedFromISR()` does the same thing. It is provided to enable application writers to implement their own version of `sbRECEIVE_COMPLETED()`, and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**

- `xMessageBuffer` — The handle of the stream buffer from which data was read.
- `pxHigherPriorityTaskWoken` — `pxHigherPriorityTaskWoken` should be initialised to `pdFALSE` before it is passed into `xMessageBufferReceiveCompletedFromISR()`. If calling `xMessageBufferReceiveCompletedFromISR()` removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then `pxHigherPriorityTaskWoken` will get set to `pdTRUE` indicating that a context switch should be performed before exiting the ISR.

**Returns**

If a task was removed from the Blocked state then `pdTRUE` is returned. Otherwise `pdFALSE` is returned.

**Type Definitions**
typedef StreamBufferHandle_t MessageBufferHandle_t

Type by which message buffers are referenced. For example, a call to xMessageBufferCreate() returns an MessageBufferHandle_t variable that can then be used as a parameter to xMessageBufferSend(), xMessageBufferReceive(), etc. Message buffer is essentially built as a stream buffer hence its handle is also set to same type as a stream buffer handle.

2.9.13 FreeRTOS (Supplemental Features)

ESP-IDF provides multiple features to supplement the features offered by FreeRTOS. These supplemental features are available on all FreeRTOS implementations supported by ESP-IDF (i.e., ESP-IDF FreeRTOS and Amazon SMP FreeRTOS). This document describes these supplemental features and is split into the following sections:

Contents

- FreeRTOS (Supplemental Features)
  - Overview
  - Ring Buffers
  - ESP-IDF Tick and Idle Hooks
  - TLSP Deletion Callbacks
  - IDF Additional API
  - Component Specific Properties
  - API Reference

Overview

ESP-IDF adds various new features to supplement the capabilities of FreeRTOS as follows:

- **Ring buffers**: Ring buffers provide a FIFO buffer that can accept entries of arbitrary lengths.
- **ESP-IDF Tick and Idle Hooks**: ESP-IDF provides multiple custom tick interrupt hooks and idle task hooks that are more numerous and more flexible when compared to FreeRTOS tick and idle hooks.
- **Thread Local Storage Pointer (TLSP) Deletion Callbacks**: TLSP Deletion callbacks are run automatically when a task is deleted, thus allowing users to clean up their TLSPs automatically.
- **IDF Additional API**: ESP-IDF specific functions added to augment the features of FreeRTOS.
- **Component Specific Properties**: Currently added only one component specific property `ORIG_INCLUDE_PATH`.

Ring Buffers

FreeRTOS provides stream buffers and message buffers as the primary mechanisms to send arbitrarily sized data between tasks and ISRs. However, FreeRTOS stream buffers and message buffers have the following limitations:

- Strictly single sender and single receiver
- Data is passed by copy
- Unable to reserve buffer space for a deferred send (i.e., send acquire)

Therefore, ESP-IDF provides a separate ring buffer implementation to address the issues above.

ESP-IDF ring buffers are strictly FIFO buffers 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.

The ring buffer provides APIs to send an item, or to allocate space for an item in the ring buffer to be filled manually by the user. For efficiency reasons, **items are always retrieved from the ring buffer by reference**. As a
result, all retrieved items must also be returned to the ring buffer by using `vRingbufferReturnItem()` or `vRingbufferReturnItemFromISR()`, 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** guarantee that an item is stored in contiguous memory and does not attempt to split an item under any circumstances. Use No-Split buffers when items must occupy contiguous memory. **Only this buffer type allows reserving buffer space for deferred sending.** Refer to the documentation of the functions `xRingbufferSendAcquire()` and `xRingbufferSendComplete()` for more details.

**Allow-Split buffers** allow an item to be split in two parts when wrapping around the end of the buffer if there is enough space at the tail and the head of the buffer combined to store the item. 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 can 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 buffers and Allow-Split buffers 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 or Allow-Split buffers requires an additional 8 bytes for a header. Item sizes are also rounded up to a 32-bit aligned size, i.e., multiple of 4 bytes. However the true item size is recorded within the header. The sizes of No-Split and 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 and then send an item to it:

```c
#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 acquire memory on the ring buffer (of type `RINGBUF_TYPE_NOSPLIT`) and then send an item to it. This adds one more step, but allows getting the address of the memory to write to, and writing to the memory yourself.

```c
#include "freertos/ringbuf.h"
#include "soc/lldesc.h"

typedef struct {
    lldesc_t dma_desc;
} (continues on next page)"
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...
    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;
    (continues on next page)
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");

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 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().

Note: Two calls to RingbufferReceive[UpTo][FromISR]() are required if the bytes wraps around the end of the ring buffer.

Sending to Ring Buffer The following diagrams illustrate the differences between No-Split and Allow-Split buffers as compared to byte buffers with regard to sending items or 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 and 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.
Fig. 42: Sending items to No-Split or Allow-Split ring buffers

However, 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. 43: 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 for the earliest acquired item prevents the subsequent items from being received.

The following diagrams illustrate what will happen when SendAcquire and SendComplete do not happen in the same order. At the beginning, there is already a 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. 44: 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. Hence, if
SendComplete is not called for the 20 bytes, it will not be available, nor will the data items following the 20 bytes item.

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 buffers and byte buffers do not allow using SendAcquire or 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 assume a buffer of 128 bytes with 56 bytes of free space that wraps around and a sent item of 28 bytes.

No-Split buffers only store an item in continuous free space and do 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.

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 treat 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 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.
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 continuous part is treated as a separate item by the byte buffer.

**Retrieving/Returning** The following diagrams illustrate the differences between No-Split and Allow-Split buffers as compared to byte buffers in retrieving and returning data:

Referring to the diagram above, the 16, 20, and 8 byte items are retrieved in FIFO order. However, the items are not returned in the order they were retrieved. First, the 20 byte item is returned followed by the 8 byte and the 16 byte items. The space is not freed until the first item, i.e., the 16 byte item is returned.
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 is 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:

```c
#include "freertos/queue.h"
#include "freertos/ringbuf.h"
...

//Create ring buffer and queue set
RingbufHandle_t buf_handle = xRingbufferCreate(1028, RINGBUF_TYPE_NOSPLIT);
QueueHandle_t queue_set = xQueueCreate(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
QueueSetMemberHandle_t 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 `xBuffSize`. Note that `xBuffSize` must be 32-bit aligned for No-Split and Allow-Split buffers.
The manner in which these blocks are allocated depends on the users requirements (e.g., all blocks being statically declared, or dynamically allocated with specific capabilities such as external RAM).

**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.

```c
#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);
```

**ESP-IDF Tick and Idle Hooks**

FreeRTOS allows applications to provide a tick hook and an idle hook at compile time:

- FreeRTOS tick hook can be enabled via the `CONFIG_FREERTOS_USE_TICK_HOOK` option. The application must provide the `vApplicationTickHook( void )` callback.
- FreeRTOS idle hook can be enabled via the `CONFIG_FREERTOS_USE_IDLE_HOOK` option. The application must provide the `vApplicationIdleHook( void )` callback.

However, the FreeRTOS tick hook and idle hook have the following draw backs:

- The FreeRTOS hooks are registered at compile time
- Only one of each hook can be registered
- On multi-core targets, the FreeRTOS hooks are symmetric, meaning each core’s tick interrupt and idle tasks ends up calling the same hook

Therefore, ESP-IDF tick and idle hooks are provided to supplement the features of FreeRTOS tick and idle hooks. The ESP-IDF hooks have the following features:

- The hooks can be registered and deregistered at run-time
- Multiple hooks can be registered (with a maximum of 8 hooks of each type per core)
- On multi-core targets, the hooks can be asymmetric, meaning different hooks can be registered to each core

ESP-IDF hooks can be registered and deregistered using the following APIs:

- For tick hooks:
Register using `esp_register_freertos_tick_hook()` or `esp_register_freertos_tick_hook_for_cpu()`.
Deregister using `esp_deregister_freertos_tick_hook()` or `esp_deregister_freertos_tick_hook_for_cpu()`.

For idle hooks:
Register using `esp_register_freertos_idle_hook()` or `esp_register_freertos_idle_hook_for_cpu()`.
Deregister using `esp_deregister_freertos_idle_hook()` or `esp_deregister_freertos_idle_hook_for_cpu()`.

Note: The tick interrupt stays active while the cache is disabled, therefore any tick hook (FreeRTOS or ESP-IDF) functions must be placed in internal RAM. Please refer to the SPI flash API documentation for more details.

### TLSP Deletion Callbacks

Vanilla FreeRTOS provides a Thread Local Storage Pointers (TLSP) feature. These are pointers stored directly in the Task Control Block (TCB) of a particular task. TLSPs allow each task to have its own unique set of pointers to data structures. Vanilla FreeRTOS expects users to:

- set a task’s TLSPs by calling `vTaskSetThreadLocalStoragePointer()` after the task has been created.
- get a task’s TLSPs by calling `pvTaskGetThreadLocalStoragePointer()` during the task’s lifetime.
- free the memory pointed to by the TLSPs before the task is deleted.

However, there can be instances where users may want the freeing of TLSP memory to be automatic. Therefore, ESP-IDF provides the additional feature of TLSP deletion callbacks. These user-provided deletion callbacks are called automatically when a task is deleted, thus allowing the TLSP memory to be cleaned up without needing to add the cleanup logic explicitly to the code of every task.

The TLSP deletion callbacks are set in a similar fashion to the TLSPs themselves.

- `vTaskSetThreadLocalStoragePointerAndDelCallback()` sets both a particular TLSP and its associated callback.
- Calling the Vanilla FreeRTOS function `vTaskSetThreadLocalStoragePointer()` simply sets the TLSP’s associated Deletion Callback to `NULL`, meaning that no callback is called for that TLSP during task deletion.

When implementing TLSP callbacks, users should note the following:

- The callback must never attempt to block or yield and critical sections should be kept as short as possible.
- The callback is called shortly before a deleted task’s memory is freed. Thus, the callback can either be called from `vTaskDelete()` itself, or from the idle task.

### IDF Additional API

The `freertos.esp_additions/include/freertos/idf_additions.h` header contains FreeRTOS-related helper functions added by ESP-IDF. Users can include this header via `#include "freertos/idf_additions.h"`.

### Component Specific Properties

Besides standard component variables that are available with basic cmake build properties, FreeRTOS component also provides arguments (only one so far) for simpler integration with other modules:

- `ORIG_INCLUDE_PATH` - contains an absolute path to freertos root include folder. Thus instead of `#include “freertos/FreeRTOS.h”` you can refer to headers directly: `#include “FreeRTOS.h”`.
Chapter 2. API Reference

API Reference

Ring Buffer API

Header File

- components/esp_ringbuf/include/freertos/ringbuf.h
- This header file can be included with:

```
#include "freertos/ringbuf.h"
```

- This header file is a part of the API provided by the `esp_ringbuf` component. To declare that your component depends on `esp_ringbuf`, add the following to your CMakeLists.txt:

```
REQUIRES esp_ringbuf
```
or

```
PRIV_REQUIRES esp_ringbuf
```

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.

**Parameters**

- `xBufferSize` - [in] Size of the buffer in bytes. Note that items require space for a header in no-split/allow-split buffers
- `xBufferType` - [in] Type of ring buffer, see documentation.

**Returns** A handle to the created ring buffer, or NULL in case of error.

**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.

**Parameters**

- `xItemSize` - [in] Size of each item to be put into the ring buffer
- `xItemNum` - [in] Maximum number of items the buffer needs to hold simultaneously

**Returns** A RingbufHandle_t handle to the created ring buffer, or NULL in case of error.

**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:** `xBufferSize` of no-split/allow-split buffers MUST be 32-bit aligned.

**Parameters**

- `xBufferSize` - [in] Size of the buffer in bytes.
- `xBufferType` - [in] Type of ring buffer, see documentation
- `pucRingbufferStorage` - [in] Pointer to the ring buffer’s storage area. Storage area must have the same size as specified by `xBufferSize`
- `pxStaticRingbuffer` - [in] Pointer to a struct of type StaticRingbuffer_t which will be used to hold the ring buffer’s data structure

**Returns** A handle to the created ring buffer
Chapter 2. API Reference

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.

**Note:** For no-split/allow-split buffers, an xItemSize of 0 will result in an item with no data being set (i.e., item only contains the header). For byte buffers, an xItemSize of 0 will simply return pdTRUE without copying any data.

**Parameters**
- `xRingbuffer` [in] Ring buffer to insert the item into
- `pvItem` [in] Pointer to data to insert. NULL is allowed if xItemSize is 0.
- `xItemSize` [in] Size of data to insert.
- `xTicksToWait` [in] Ticks to wait for room in the ring buffer.

**Returns**
- pdTRUE if succeeded
- pdFALSE on time-out or when the data is larger than the maximum permissible size of the 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.

**Note:** For no-split/allow-split buffers, an xItemSize of 0 will result in an item with no data being set (i.e., item only contains the header). For byte buffers, an xItemSize of 0 will simply return pdTRUE without copying any data.

**Parameters**
- `xRingbuffer` [in] Ring buffer to insert the item into
- `pvItem` [in] Pointer to data to insert. NULL is allowed if xItemSize is 0.
- `xItemSize` [in] Size of data to insert.
- `pxHigherPriorityTaskWoken` [out] Value pointed to will be set to pdTRUE if the function woke up a higher priority task.

**Returns**
- pdTRUE if succeeded
- pdFALSE when the ring buffer does not have space.

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 times out.

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.

**Note:** An xItemSize of 0 will result in a buffer being acquired, but the buffer will have a size of 0.

### Parameters
- **xRingbuffer** – [in] Ring buffer to allocate the memory
- **ppvItem** – [out] Double pointer to memory acquired (set to NULL if no memory were retrieved)
- **xItemSize** – [in] Size of item to acquire.
- **xTicksToWait** – [in] Ticks to wait for room in the ring buffer.

### Returns
- pdTRUE if succeeded
- pdFALSE on time-out or when the data is larger than the maximum permissible size of the 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.

### Parameters
- **xRingbuffer** – [in] Ring buffer to insert the item into
- **pvItem** – [in] Pointer to item in allocated memory to insert.

### Returns
- pdTRUE if succeeded
- pdFALSE if fail for some reason.

```
void *xRingbufferReceive(RingbufHandle_t xRingbuffer, size_t *pxItemSize, TickType_t xTicksToWait)
```

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.

**Note:** It is possible to receive items with a pxItemSize of 0 on no-split/allow split buffers.

### Parameters
- **xRingbuffer** – [in] Ring buffer to retrieve the item from
- **pxItemSize** – [out] Pointer to a variable to which the size of the retrieved item will be written.
- **xTicksToWait** – [in] 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.

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.

Note: Two calls to RingbufferReceiveFromISR() are required if the bytes wrap around the end of the ring buffer.

Note: It is possible to receive items with a pxItemSize of 0 on no-split/allow split buffers.

Parameters
• xRingbuffer [in] Ring buffer to retrieve the item from
• pxItemSize [out] Pointer to a variable to which the size of the retrieved item will be written.

Returns
• Pointer to the retrieved item on success; *pxItemSize filled with the length of the item.
• NULL on timeout, *pxItemSize is untouched in that case.

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.

Note: It is possible to receive items with a pxItemSize of 0 on allow split buffers.

Parameters
• xRingbuffer –[in] Ring buffer to retrieve the item from
• ppvHeadItem –[out] Double pointer to first part (set to NULL if no items were retrieved)
• ppvTailItem –[out] Double pointer to second part (set to NULL if item is not split)
Chapter 2. API Reference

- **pxHeadItemSize** - [out] Pointer to size of first part (unmodified if no items were retrieved)
- **pxTailItemSize** - [out] Pointer to size of second part (unmodified if item is not split)
- **xTicksToWait** - [in] Ticks to wait for items in the ring buffer.

**Returns**
- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

```c
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

**Note:** It is possible to receive items with a pxItemSize of 0 on allow split buffers.

**Parameters**
- **xRingbuffer** - [in] Ring buffer to retrieve the item from
- **ppvHeadItem** - [out] Double pointer to first part (set to NULL if no items were retrieved)
- **ppvTailItem** - [out] Double pointer to second part (set to NULL if item is not split)
- **pxHeadItemSize** - [out] Pointer to size of first part (unmodified if no items were retrieved)
- **pxTailItemSize** - [out] Pointer to size of second part (unmodified if item is not split)

**Returns**
- pdTRUE if an item (split or unsplit) was retrieved
- pdFALSE when no item was retrieved

```c
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
Chapter 2. API Reference

**Note:** Two calls to `RingbufferReceiveUpTo()` are required if the bytes wrap around the end of the ring buffer.

**Parameters**
- `xRingbuffer` [in] Ring buffer to retrieve the item from
- `pxItemSize` [out] Pointer to a variable to which the size of the retrieved item will be written.
- `xTicksToWait` [in] Ticks to wait for items in the ring buffer.
- `xMaxSize` [in] Maximum number of bytes to return.

**Returns**
- Pointer to the retrieved item on success; `*pxItemSize` filled with the length of the item.
- NULL on timeout, `*pxItemSize` is untouched in that case.

```c
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

**Parameters**
- `xRingbuffer` [in] Ring buffer to retrieve the item from
- `pxItemSize` [out] Pointer to a variable to which the size of the retrieved item will be written.
- `xMaxSize` [in] Maximum number of bytes to return. Size of 0 simply returns NULL.

**Returns**
- 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.

```c
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**
- `xRingbuffer` [in] Ring buffer the item was retrieved from
- `pvItem` [in] Item that was received earlier

```c
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

- `xRingbuffer` [in] Ring buffer the item was retrieved from
- `pvItem` [in] Item that was received earlier
- `pxHigherPriorityTaskWoken` [out] Value pointed to will be set to `pdTRUE` if the function woke up a higher priority task.

```c
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 be the user.

```c
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 `((buffer_size/2) - header_size)`. This limit is imposed so that an item of max item size can always be sent to an empty no-split buffer regardless of the internal positions of the buffer’s read/write/free pointers.

```c
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.

**Note:** An empty no-split buffer has a max current free size for an item that is limited to `((buffer_size/2) - header_size)`. See API reference for `xRingbufferGetMaxItemSize()`.

**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

```c
BaseType_t xRingbufferAddToQueueSetRead (RingbufHandle_t xRingbuffer, QueueSetHandle_t xQueueSet)
```
Add the ring buffer to a queue set. Notified when data has been written to the ring buffer.

This function adds the ring buffer to a queue set, thus allowing a task to block on multiple queues/ring buffers. The queue set is notified when the new data becomes available to read on the ring buffer.

**Parameters**

- `xRingbuffer` [in] Ring buffer to add to the queue set
- `xQueueSet` [in] Queue set to add the ring buffer to

**Returns**

- `pdTRUE` on success, `pdFALSE` otherwise
static inline BaseType_t xRingbufferCanRead (RingbufHandle_t xRingbuffer, QueueSetMemberHandle_t xMember)  

Check if the selected queue set member is a particular ring buffer.  

This API checks if queue set member returned from xQueueSelectFromSet() is a particular ring buffer. If so, this indicates the ring buffer has items waiting to be retrieved.

Parameters
• xRingbuffer –[in] Ring buffer to check
• xMember –[in] Member returned from xQueueSelectFromSet

Returns
• pdTRUE when selected queue set member is the ring buffer
• pdFALSE otherwise.

BaseType_t xRingbufferRemoveFromQueueSetRead (RingbufHandle_t xRingbuffer, QueueSetHandle_t xQueueSet)  

Remove the ring buffer from a queue set.

This function removes a ring buffer from a queue set. The ring buffer must have been previously added to the queue set using xRingbufferAddToQueueSetRead().

Parameters
• xRingbuffer –[in] Ring buffer to remove from the queue set
• xQueueSet –[in] Queue set to remove the ring buffer from

Returns
• pdTRUE on success
• pdFALSE otherwise

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 a ring buffer’s current status such as free/read/write/acquire pointer positions, and number of items waiting to be retrieved. Arguments can be set to NULL if they are not required.

Parameters
• xRingbuffer –[in] Ring buffer to remove from the queue set
• uxFree –[out] Pointer use to store free pointer position
• uxRead –[out] Pointer use to store read pointer position
• uxWrite –[out] Pointer use to store write pointer position
• uxAcquire –[out] Pointer use to store acquire pointer position
• uxItemsWaiting –[out] 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

BaseType_t xRingbufferGetStaticBuffer (RingbufHandle_t xRingbuffer, uint8_t **ppucRingbufferStorage, StaticRingbuffer_t **ppxStaticRingbuffer)  

Retrieve the pointers to a statically created ring buffer.

Parameters
• xRingbuffer –[in] Ring buffer
• ppucRingbufferStorage –[out] Used to return a pointer to the queue’s storage area buffer
• ppxStaticRingbuffer –[out] Used to return a pointer to the queue’s data structure buffer

Returns
• pdTRUE if buffers were retrieved, pdFALSE otherwise.
**RingbufHandle_t xRingbufferCreateWithCaps** (size_t xBufferSize, RingbufferType_t xBufferType, UBaseType_t uxMemoryCaps)

Creates a ring buffer with specific memory capabilities.

This function is similar to xRingbufferCreate(), except that it allows the memory allocated for the ring buffer to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** A queue created using this function must only be deleted using vRingbufferDeleteWithCaps()

**Parameters**

- **xBufferSize** - [in] Size of the buffer in bytes
- **xBuffertype** - [in] Type of ring buffer, see documentation.
- **uxMemoryCaps** - [in] Memory capabilities of the queue’s memory (see esp_heap_caps.h)

**Returns** Handle to the created ring buffer or NULL on failure.

```c
void vRingbufferDeleteWithCaps(RingbufHandle_t xRingbuffer)
```

Deletes a ring buffer previously created using xRingbufferCreateWithCaps()

**Parameters** `xRingbuffer` - Ring buffer

**Structures**

```c
define 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.

**Type Definitions**

```c
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.

```c
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.

**Enumerations**

```c
define RingbufferType_t
```

Values:

- **RINGBUF_TYPE_NOSPLIT**

  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.
enumerator **RINGBUF_TYPE_ALLOW_SPLIT**

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.

enumerator **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.

enumerator **RINGBUF_TYPE_MAX**

Hooks API

Header File

- components/esp_system/include/esp_freertos_hooks.h
- This header file can be included with:

  ```c
  #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.

**Parameters**

- **new_idle_cb** — [in] Callback to be called
- **cpuid** — [in] id of the core

**Returns**

- 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

**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.

**Parameters**

- **new_idle.cb** — [in] Callback to be called

**Returns**

- 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
**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.

**Parameters**
- new_tick_cb -[in] Callback to be called
- cpuid -[in] id of the core

**Returns**
- 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

**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.

**Parameters**
- new_tick_cb -[in] Callback to be called

**Returns**
- 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

**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**
- old_idle_cb -[in] Callback to be unregistered
- cpuid -[in] 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**
- old_idle_cb -[in] 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**
- old_tick_cb -[in] Callback to be unregistered
- cpuid -[in] 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**
- old_tick_cb -[in] Callback to be unregistered

**Type Definitions**

typedef bool (*esp_freertos_idle_cb_t)(void)

typedef void (*esp_freertos_tick_cb_t)(void)

**Additional API**
Header File

- components/freertos/esp_additions/include/freertos/idf_additions.h
- This header file can be included with:

```c
#include "freertos/idf_additions.h"
```

Functions

```c
BaseType_t xTaskCreatePinnedToCore(TaskFunction_t pxTaskCode, const char *const pcName, const uint32_t ulStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pxCreatedTask, const BaseType_t xCoreID)
```

Create a new task that is pinned to a particular core.

This function is similar to xTaskCreate(), but allows the creation of a pinned task. The task’s pinned core is specified by the xCoreID argument. If xCoreID is set to tskNO_AFFINITY, then the task is unpinned and can run on any core.

**Note:** If (configNUM_CORES == 1), xCoreID is ignored.

**Parameters**

- **pxTaskCode** - Pointer to the task entry function.
- **pcName** - A descriptive name for the task.
- **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 should run.
- **pxCreatedTask** - Used to pass back a handle by which the created task can be referenced.
- **xCoreID** - The core to which the task is pinned to, or tskNO_AFFINITY if the task has no core affinity

**Returns**

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

```c
TaskHandle_t xTaskCreateStaticPinnedToCore(TaskFunction_t pxTaskCode, const char *const pcName, const uint32_t ulStackDepth, void *const pvParameters, UBaseType_t uxPriority, StackType_t *const puxStackBuffer, StaticTask_t *const pxTaskBuffer, const BaseType_t xCoreID)
```

Create a new static task that is pinned to a particular core.

This function is similar to xTaskCreateStatic(), but allows the creation of a pinned task. The task’s pinned core is specified by the xCoreID argument. If xCoreID is set to tskNO_AFFINITY, then the task is unpinned and can run on any core.

**Note:** If (configNUM_CORES == 1), xCoreID is ignored.

**Parameters**

- **pxTaskCode** - Pointer to the task entry function.
- **pcName** - A descriptive name for the task.
- **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 should run.
- **puxStackBuffer** - Must point to a StackType_t array that has at least ulStackDepth indexes
Chapter 2  API Reference

*pxTaskBuffer* - Must point to a variable of type StaticTask_t, which will then be used to hold the task’s data structures,

*xCoreID* - The core to which the task is pinned to, or tskNO_AFFINITY if the task has no core affinity

**Returns** The task handle if the task was created, NULL otherwise.

```c
BaseType_t xTaskGetCoreID (TaskHandle_t xTask)
```

Get the current core ID of a particular task.

Helper function to get the core ID of a particular task. If the task is pinned to a particular core, the core ID is returned. If the task is not pinned to a particular core, tskNO_AFFINITY is returned.

If CONFIG_FREERTOS_UNICORE is enabled, this function simply returns 0.

[refactor-todo] See if this needs to be deprecated (IDF-8145)(IDF-8164)

**Note:** If CONFIG_FREERTOS_SMP is enabled, please call vTaskCoreAffinityGet() instead.

**Note:** In IDF FreerTOS when configNUMBER_OF_CORES == 1, this function will always return 0.

**Parameters**

- **xTask** - The task to query

**Returns** The task’s core ID or tskNO_AFFINITY

```c
TaskHandle_t xTaskGetIdleTaskHandleForCore (BaseType_t xCoreID)
```

Get the handle of idle task for the given core.

[refactor-todo] See if this needs to be deprecated (IDF-8145)

**Note:** If CONFIG_FREERTOS_SMP is enabled, please call xTaskGetIdleTaskHandle() instead.

**Parameters**

- **xCoreID** - The core to query

**Returns** Handle of the idle task for the queried core

```c
TaskHandle_t xTaskGetCurrentTaskHandleForCore (BaseType_t xCoreID)
```

Get the handle of the task currently running on a certain core.

Because of the nature of SMP processing, there is no guarantee that this value will still be valid on return and should only be used for debugging purposes.

[refactor-todo] See if this needs to be deprecated (IDF-8145)

**Note:** If CONFIG_FREERTOS_SMP is enabled, please call xTaskGetCurrentTaskHandleCPU() instead.

**Parameters**

- **xCoreID** - The core to query

**Returns** Handle of the current task running on the queried core

```c
uint8_t* pxTaskGetStackStart (TaskHandle_t xTask)
```

Returns the start of the stack associated with xTask.

Returns the lowest stack memory address, regardless of whether the stack grows up or down.

[refactor-todo] Change return type to StackType_t (IDF-8158)

**Parameters**

- **xTask** - Handle of the task associated with the stack returned. Set xTask to NULL to return the stack of the calling task.

**Returns** A pointer to the start of the stack.
void **TaskSetThreadLocalStoragePointerAndDelCallback**(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 **xTaskCreatePinnedToCoreWithCaps**(TaskFunction_t pvTaskCode, const char *const pcName, configSTACK_DEPTH_TYPE usStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pvCreatedTask, const BaseType_t xCoreID, UBaseType_t uxMemoryCaps)

Creates a pinned task where its stack has specific memory capabilities.

This function is similar to xTaskCreatePinnedToCore(), except that it allows the memory allocated for the task’s stack to have specific capabilities (e.g., MALLOC_CAP_SPIRAM).

However, the specified capabilities will NOT apply to the task’s TCB as a TCB must always be in internal RAM.

**Parameters**
- pvTaskCode: Pointer to the task entry function
- pcName: A descriptive name for the task
- usStackDepth: The size of the task stack specified as the number of bytes
- pvParameters: Pointer that will be used as the parameter for the task being created.
- uxPriority: The priority at which the task should run.
- pvCreatedTask: Used to pass back a handle by which the created task can be referenced.
- xCoreID: Core to which the task is pinned to, or tskNO_AFFINITY if unpinned.
- uxMemoryCaps: Memory capabilities of the task stack’s memory (see esp_heap_caps.h)

**Returns**
- pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

static inline BaseType_t **xTaskCreateWithCaps**(TaskFunction_t pvTaskCode, const char *const pcName,
configSTACK_DEPTH_TYPE usStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *pvCreatedTask, UBaseType_t uxMemoryCaps)

Creates a task where its stack has specific memory capabilities.

This function is similar to xTaskCreate(), except that it allows the memory allocated for the task’s stack to have specific capabilities (e.g., MALLOC_CAP_SPIRAM).

However, the specified capabilities will NOT apply to the task’s TCB as a TCB must always be in internal RAM.
Chapter 2. API Reference

Note: A task created using this function must only be deleted using vTaskDeleteWithCaps()

Parameters

- **pvTaskCode**: Pointer to the task entry function
- **pcName**: A descriptive name for the task
- **usStackDepth**: The size of the task stack specified as the number of bytes
- **pvParameters**: Pointer that will be used as the parameter for the task being created.
- **uxPriority**: The priority at which the task should run.
- **pvCreatedTask**: Used to pass back a handle by which the created task can be referenced.
- **uxMemoryCaps**: Memory capabilities of the task stack’s memory (see esp_heap_caps.h)

Returns: pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

void vTaskDeleteWithCaps (TaskHandle_t xTaskToDelete)

Deletes a task previously created using xTaskCreateWithCaps() or xTaskCreatePinnedToCoreWithCaps()

Parameters

- **xTaskToDelete**: A handle to the task to be deleted

Note: A queue created using this function must only be deleted using vQueueDeleteWithCaps()

Parameters

- **uxQueueLength**: The maximum number of items that the queue can contain.
- **uxItemSize**: The number of bytes each item in the queue will require.
- **uxMemoryCaps**: Memory capabilities of the queue’s memory (see esp_heap_caps.h)

Returns: Handle to the created queue or NULL on failure.

void vQueueDeleteWithCaps (QueueHandle_t xQueue)

Deletes a queue previously created using xQueueCreateWithCaps()

Parameters

- **xQueue**: A handle to the queue to be deleted.

Note: A binary semaphore created using this function must only be deleted using vSemaphoreDeleteWithCaps()

Parameters

- **uxMemoryCaps**: Memory capabilities of the binary semaphore’s memory (see esp_heap_caps.h)

Returns: Handle to the created binary semaphore or NULL on failure.
static inline SemaphoreHandle_t xSemaphoreCreateCountingWithCaps (UBaseType_t uxMaxCount,
                                             UBaseType_t uxInitialCount,
                                             UBaseType_t uxMemoryCaps)

Creates a counting semaphore with specific memory capabilities.

This function is similar to xSemaphoreCreateCounting(), except that it allows the memory allocated for the
counting semaphore to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** A counting semaphore created using this function must only be deleted using vSemaphoreDeleteWithCaps()

**Parameters**
- **uxMaxCount** - The maximum count value that can be reached.
- **uxInitialCount** - The count value assigned to the semaphore when it is created.
- **uxMemoryCaps** - Memory capabilities of the counting semaphore’s memory (see esp_heap_caps.h)

**Returns** Handle to the created counting semaphore or NULL on failure.

static inline SemaphoreHandle_t xSemaphoreCreateMutexWithCaps (UBaseType_t uxMemoryCaps)

Creates a mutex semaphore with specific memory capabilities.

This function is similar to xSemaphoreCreateMutex(), except that it allows the memory allocated for the mutex
semaphore to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** A mutex semaphore created using this function must only be deleted using vSemaphoreDeleteWithCaps()

**Parameters** **uxMemoryCaps** - Memory capabilities of the mutex semaphore’s memory (see esp_heap_caps.h)

**Returns** Handle to the created mutex semaphore or NULL on failure.

static inline SemaphoreHandle_t xSemaphoreCreateRecursiveMutexWithCaps (UBaseType_t
                                                                         uxMemoryCaps)

Creates a recursive mutex with specific memory capabilities.

This function is similar to xSemaphoreCreateRecursiveMutex(), except that it allows the memory allocated for
the recursive mutex to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** A recursive mutex created using this function must only be deleted using vSemaphoreDeleteWithCaps()

**Parameters** **uxMemoryCaps** - Memory capabilities of the recursive mutex’s memory (see esp_heap_caps.h)

**Returns** Handle to the created recursive mutex or NULL on failure.

void vSemaphoreDeleteWithCaps (SemaphoreHandle_t xSemaphore)

Deletes a semaphore previously created using one of the xSemaphoreCreate…WithCaps() functions.

**Parameters** **xSemaphore** - A handle to the semaphore to be deleted.

static inline StreamBufferHandle_t xStreamBufferCreateWithCaps (size_t xBufferSizeBytes,
                                                                size_t
                                                                xTriggerLevelBytes,
                                                                UBaseType_t
                                                                uxMemoryCaps)

Creates a stream buffer with specific memory capabilities.
This function is similar to xStreamBufferCreate(), except that it allows the memory allocated for the stream buffer to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** A stream buffer created using this function must only be deleted using vStreamBufferDeleteWithCaps()

### Parameters
- **xBufferSizeBytes** - The total number of bytes the stream buffer will be able to hold at any one time.
- **xTriggerLevelBytes** - The number of bytes that must be in the stream buffer before unblocking
- **uxMemoryCaps** - Memory capabilities of the stream buffer’s memory (see esp_heap_caps.h)

### Returns
Handle to the created stream buffer or NULL on failure.

```c
static inline void vStreamBufferDeleteWithCaps(StreamBufferHandle_t xStreamBuffer)
```

Deletes a stream buffer previously created using xStreamBufferCreateWithCaps()

**Parameters**
- **xStreamBuffer** - A handle to the stream buffer to be deleted.

### Parameters
- **xBufferSizeBytes** - The total number of bytes (not messages) the message buffer will be able to hold at any one time.
- **uxMemoryCaps** - Memory capabilities of the message buffer’s memory (see esp_heap_caps.h)

### Returns
Handle to the created message buffer or NULL on failure.

```c
static inline MessageBufferHandle_t xMessageBufferCreateWithCaps(size_t xBufferSizeBytes, UBaseType_t uxMemoryCaps)
```

Creates a message buffer with specific memory capabilities.

This function is similar to xMessageBufferCreate(), except that it allows the memory allocated for the message buffer to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** A message buffer created using this function must only be deleted using vMessageBufferDeleteWithCaps()

### Parameters
- **xBufferSizeBytes** - The total number of bytes (not messages) the message buffer will be able to hold at any one time.
- **uxMemoryCaps** - Memory capabilities of the message buffer’s memory (see esp_heap_caps.h)

### Returns
Handle to the created message buffer or NULL on failure.

```c
static inline void vMessageBufferDeleteWithCaps(MessageBufferHandle_t xMessageBuffer)
```

Deletes a stream buffer previously created using xMessageBufferCreateWithCaps()

**Parameters**
- **xMessageBuffer** - A handle to the message buffer to be deleted.

### Parameters
- **uxMemoryCaps** - Memory capabilities of the event group’s memory (see esp_heap_caps.h)

### Returns
Handle to the created event group or NULL on failure.

```c
EventGroupHandle_t xEventGroupCreateWithCaps(UBaseType_t uxMemoryCaps)
```

Creates an event group with specific memory capabilities.

This function is similar to xEventGroupCreate(), except that it allows the memory allocated for the event group to have specific capabilities (e.g., MALLOC_CAP_INTERNAL).

**Note:** An event group created using this function must only be deleted using vEventGroupDeleteWithCaps()
Chapter 2. API Reference

void vEventGroupDeleteWithCaps(EventGroupHandle_t xEventGroup)

Deletes an event group previously created using xEventGroupCreateWithCaps()

**Parameters**
- **xEventGroup**: A handle to the event group to be deleted.

**Type Definitions**
typedef void (*TlsDeleteCallbackFunction_t)(int, void*)

Prototype of local storage pointer deletion callback.

### 2.9.14 Heap Memory Allocation

**Stack and Heap**

ESP-IDF applications use the common computer architecture patterns of **stack** (dynamic memory allocated by program control flow), **heap** (dynamic memory allocated by function calls), and **static memory** (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-S2 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 C Standard Library’s `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 a memory with certain properties (e.g., 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-S2 contains multiple types of RAM:

- **DRAM (Data RAM)** is memory that is connected to CPU’s data bus and is used to hold data. This is the most common kind of memory accessed as a heap.
- **IRAM (Instruction RAM)** is memory that is connected to the CPU’s instruction bus and usually holds executable data only (i.e., instructions). If accessed as generic memory, all accesses must be aligned to 32-Bit Accessible Memory.
- **D/IRAM** is RAM that is connected to CPU’s data bus and instruction bus, thus can be used either Instruction or Data RAM.

For more details on these internal memory types, see **Memory Types**.

It is also possible to connect external SPI RAM to the ESP32-S2. The **external RAM** is integrated into the ESP32-S2’s memory map via the cache, and accessed similarly to DRAM.

All DRAM memory is single-byte accessible, thus all DRAM heaps possess the `MALLOC_CAP_8BIT` capability. Users can call `heap_caps_get_free_size(MALLOC_CAP_8BIT)` to get the free size of all DRAM heaps.

If ran out of `MALLOC_CAP_8BIT`, the users can use `MALLOC_CAP_IRAM_8BIT` instead. In that case, IRAM can still be used as a “reserve” pool of internal memory if the users only access it in a 32-bit aligned manner, or if they enable `CONFIG_ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY`.

When calling `malloc()`, the ESP-IDF `malloc()` internally calls `heap_caps_malloc_default(size)`. This will allocate memory with the capability `MALLOC_CAP_DEFAULT`, which is byte-addressable.
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 that is not statically allocated by the app. Reducing statically-allocated buffers increases the amount of available free heap.

To find the amount of statically allocated memory, use the `idf.py size` command.

**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 that 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-S2 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 3FFA6E0 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, which is sometimes unavailable for a normal `malloc()` call. This can help to use all the available memory in the ESP32-S2.

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 4 MiB in size can be allocated using standard `malloc` calls, or via `heap_caps_malloc(MALLOC_CAP_SPIRAM)`, depending on the configuration. See Configuring External RAM for more details.
## 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 (see *Calling Heap-Related Functions from ISR*). 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.

### Calling Heap-Related Functions from ISR

The following functions from the heap component can be called from the interrupt handler (ISR):

- `heap_caps_malloc()`
- `heap_caps_malloc_default()`
- `heap_caps_realloc_default()`
- `heap_caps_malloc_prefer()`
- `heap_caps_realloc_prefer()`
- `heap_caps_calloc_prefer()`
- `heap_caps_free()`
- `heap_caps_realloc()`
- `heap_caps_calloc()`
- `heap_caps_aligned_alloc()`
- `heap_caps_aligned_free()`

**Note:** However, this practice is strongly discouraged.

### Heap Tracing & Debugging

The following features are documented on the *Heap Memory Debugging* page:

- *Heap Information* (free space, etc.)
- *Heap Allocation and Free Function Hooks*
- *Heap Corruption Detection*
- *Heap Tracing* (memory leak detection, monitoring, etc.)

### 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 prioritized, so that (for example) dedicated DRAM and IRAM regions are 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()` for the heap situated in that particular region.

Calling `free()` involves finding the particular heap corresponding to the freed address, and then call `multi_heap_free()` on that particular `multi_heap` instance.
API Reference - Heap Allocation

Header File

- components/heap/include/esp_heap_caps.h
- This header file can be included with:

```c
#include "esp_heap_caps.h"
```

Functions

**esp_err_t** `heap_caps_register_failed_alloc_callback` *(esp_alloc_failed_hook_t callback)*

- registers a callback function to be invoked if a memory allocation operation fails

  **Parameters**
  - `callback` - caller defined callback to be invoked

  **Returns**
  - ESP_OK if callback was registered.

**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.**

  **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

  **Returns**
  - A pointer to the memory allocated on success, NULL on failure

**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, uint32_t 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.

  **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.

  **Returns**
  - Pointer to a new buffer of size ‘size’ with capabilities ‘caps’, or NULL if allocation failed.

**void** `heap_caps_aligned_alloc` *(size_t alignment, size_t size, uint32_t caps)*

- Allocate an aligned chunk of memory which has the given capabilities.

  **Equivalent semantics to libc aligned_alloc(), for capability-aware memory.**

  **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
Returns A pointer to the memory allocated on success, NULL on failure

void heap_caps_aligned_free (void *ptr)
Used to deallocate memory previously allocated with heap_caps_aligned_alloc.

Note: This function is deprecated, please consider using heap_caps_free() instead

Parameters ptr - Pointer to the memory allocated

void *heap_caps_aligned_calloc (size_t alignment, size_t n, size_t size, uint32_t caps)
Allocate an aligned chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

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

Returns A pointer to the memory allocated on success, NULL on failure

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 malloc(), for capability-aware memory.

In IDF, calloc(p) is equivalent to heap_caps_calloc(p, MALLOC_CAP_8BIT).

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

Returns A pointer to the memory allocated on success, NULL on failure

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.

Parameters caps – Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

Returns total size in bytes

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: 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.

Parameters caps – Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

Returns Amount of free bytes in the regions
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 watermarks of the regions capable of delivering the memory with the given capabilities.

**Note:** Note the result may be less than the global all-time minimum available heap of this kind, as “low watermarks” are tracked per-region. Individual regions’ heaps may have reached their “low watermarks” at different points in time. However, this result still gives a “worst case” indication for all-time minimum free heap.

**Parameters** caps – Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

**Returns** Amount of free bytes in the regions

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.

**Parameters** caps – Bitwise OR of MALLOC_CAP_* flags indicating the type of memory

**Returns** Size of the largest free block in bytes.

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.

**Note:** Please increase the value of CONFIG_ESP_INT_WDT_TIMEOUT_MS when using this API with PSRAM enabled.

**Parameters** print_errors – Print specific errors if heap corruption is found.

**Returns** True if all heaps are valid, False if at least one heap is corrupt.

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.

**Note:** Please increase the value of `CONFIG_ESP_INT_WDT_TIMEOUT_MS` when using this API with PSRAM capability flag.

### Parameters
- **caps** - Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory
- **print_errors** - Print specific errors if heap corruption is found.

### Returns
True if all heaps are valid, False if at least one heap is corrupt.

```c
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.

### Parameters
- **addr** - Address in memory. Check for corruption in region containing this address.
- **print_errors** - Print specific errors if heap corruption is found.

### Returns
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.

```c
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.

```c
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.

### Parameters
- **size** - Size, in bytes, of the amount of memory to allocate
- **num** - Number of variable parameters

### Returns
A pointer to the memory allocated on success, NULL on failure

```c
void *heap_caps_realloc_prefer (void *ptr, size_t size, size_t num, ...)
```

Reallocate a chunk of memory as preference in decreasing order.

### Parameters
Chapter 2. API Reference

- **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 parameters

**Returns**
Point to a new buffer of size ‘size’, or NULL if allocation failed.

```c
void *heap_caps_calloc_prefer(size_t n, size_t size, size_t num, ...)
```
Allocate a chunk of memory as preference in decreasing order.

**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 parameters

**Returns**
A pointer to the memory allocated on success, NULL on failure

```c
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

```c
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.

```c
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.

**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.

**Returns**
Size of the memory allocated at this block.

**Macros**

**HEAP_IRAM_ATTR**

**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 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_IRAM_8BIT
Memory must be in IRAM and allow unaligned access.

MALLOC_CAP_RETENTION
Memory must be able to accessed by retention DMA.

MALLOC_CAP_RTCRAM
Memory must be in RTC fast memory.

MALLOC_CAP_TCM
Memory must be in TCM memory.

MALLOC_CAP_INVALID
Memory can’t be used / list end marker.
Chapter 2. API Reference

Type Definitions

typedef void (*esp_alloc_failed_hook_t)(size_t size, uint32_t caps, const char *function_name)
callback called when an allocation operation fails, if registered

- **Param size**: in bytes of failed allocation
- **Param caps**: capabilities requested of failed allocation
- **Param function_name**: function which generated the failure

API Reference - Initialisation

Header File

- components/heap/include/esp_heap_caps_init.h
- This header file can be included with:

```c
#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.

**Note:** Please refer to following example for memory regions allowed for addition to heap based on an existing region (address range for demonstration purpose only):

```
Existing region: 0x1000 <-> 0x3000
New region: 0x1000 <-> 0x3000 (Allowed)
New region: 0x1000 <-> 0x2000 (Allowed)
New region: 0x0000 <-> 0x1000 (Allowed)
New region: 0x3000 <-> 0x4000 (Allowed)
New region: 0x0000 <-> 0x2000 (NOT Allowed)
New region: 0x0000 <-> 0x4000 (NOT Allowed)
New region: 0x1000 <-> 0x4000 (NOT Allowed)
New region: 0x2000 <-> 0x4000 (NOT Allowed)
```
Parameters

- **start** – Start address of new region.
- **end** – End address of new region.

Returns 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().

```c
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.

**Note:** Please refer to following example for memory regions allowed for addition to heap based on an existing region (address range for demonstration purpose only):

| Existing region: 0x1000 <-> 0x3000 |
|-------------------------|----------|
| New region: 0x1000 <-> 0x3000 (Allowed) |
| New region: 0x1000 <-> 0x2000 (Allowed) |
| New region: 0x0000 <-> 0x1000 (Allowed) |
| New region: 0x3000 <-> 0x4000 (Allowed) |
| New region: 0x0000 <-> 0x2000 (NOT Allowed) |
| New region: 0x0000 <-> 0x4000 (NOT Allowed) |
| New region: 0x1000 <-> 0x4000 (NOT Allowed) |
| New region: 0x2000 <-> 0x4000 (NOT Allowed) |

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.

Returns

- 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

**API Reference - Multi-Heap API**

(Note: The multi-heap API is used internally by the heap capabilities allocator. Most ESP-IDF programs never need to call this API directly.)

**Header File**

- components/heap/include/multi_heap.h
- This header file can be included with:

```c
#include "multi_heap.h"
```

**Functions**

```c
void *multi_heap_aligned_alloc (multi_heap_handle_t heap, size_t size, size_t alignment)
```

allocate a chunk of memory with specific alignment

**Parameters**

- **heap** – Handle to a registered heap.
• **size** - size in bytes of memory chunk
• **alignment** - how the memory must be aligned

**Returns**
pointer to the memory allocated, NULL on failure

```c
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.

**Parameters**
• **heap** - Handle to a registered heap.
• **size** - Size of desired buffer.

**Returns**
Pointer to new memory, or NULL if allocation fails.

```c
void multi_heap_aligned_free (multi_heap_handle_t heap, void *p)
```

free() a buffer aligned in a given heap.

**Note:** This function is deprecated, consider using multi_heap_free() instead

```c
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_aligned_alloc() for the same heap.

```c
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.

**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.

**Returns**
New buffer of ‘size’ containing contents of ‘p’, or NULL if reallocation failed.

```c
size_t multi_heap_get_allocated_size (multi_heap_handle_t heap, void *p)
```

Return the size that a particular pointer was allocated with.

**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.

**Returns**
Size of the memory allocated at this block. May be more than the original size argument, due to padding and minimum block sizes.

```c
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.
Chapter 2. API Reference

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.

**Parameters**

- start - Start address of the memory to use for a new heap.
- size - Size (in bytes) of the new heap.

**Returns** Handle of a new heap ready for use, or NULL if the heap region was too small to be initialised.

```c
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.

```c
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.

```c
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.

**Note:** This function is not thread-safe as it sets a global variable with the value of print_errors.

```c
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().

**Parameters** heap - Handle to a registered heap.

**Returns** Number of free bytes.

```c
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 watermark” of possible values returned from multi_free_heap_size(), for the specified heap.
Parameters heap - Handle to a registered heap.
Returns Number of free bytes.

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.

void *multi_heap_aligned_alloc_offs (multi_heap_handle_t heap, size_t size, size_t alignment, size_t offset)
Perform an aligned allocation from the provided offset.

Parameters
• heap - The heap in which to perform the allocation
• size - The size of the allocation
• alignment - How the memory must be aligned
• offset - The offset at which the alignment should start
Returns void* The ptr to the allocated memory

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 the 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.
2.9.15 Memory Management for MMU Supported Memory

Introduction

ESP32-S2 Memory Management Unit (MMU) is relatively simple. It can do memory address translation between physical memory addresses and virtual memory addresses. So CPU can access physical memories via virtual addresses. There are multiple types of virtual memory addresses, which have different capabilities.

ESP-IDF provides a memory mapping driver that manages the relation between these physical memory addresses and virtual memory addresses, so as to achieve some features such as reading from SPI flash via a pointer.

Memory mapping driver is actually a capabilities-based virtual memory address allocator that allows applications to make virtual memory address allocations for different purposes. In the following chapters, we call this driver esp_mmap driver.

ESP-IDF also provides a memory synchronization driver which can be used for potential memory desynchronization scenarios.

Physical Memory Types

Memory mapping driver currently supports mapping to following physical memory type(s):

- SPI flash
- PSRAM

Virtual Memory Capabilities

- MMU_MEM_CAP_EXEC: This capability indicates that the virtual memory address has the execute permission. Note this permission scope is within the MMU hardware.
- MMU_MEM_CAP_READ: This capability indicates that the virtual memory address has the read permission. Note this permission scope is within the MMU hardware.
- MMU_MEM_CAP_WRITE: This capability indicates that the virtual memory address has the write permission. Note this permission scope is within the MMU hardware.
- MMU_MEM_CAP_32BIT: This capability indicates that the virtual memory address allows for 32 bits or multiples of 32 bits access.
- MMU_MEM_CAP_8BIT: This capability indicates that the virtual memory address allows for 8 bits or multiples of 8 bits access.

4 MB external memory addresses (from 0x40400000 to 0x40800000) which have the MMU_MEM_CAP_EXEC and MMU_MEM_CAP_READ capabilities are not available for users to allocate, due to hardware limitations.

You can call esp_mmu_map_get_max_consecutive_free_block_size() to know the largest consecutive mappable block size with certain capabilities.

Memory Management Drivers

Driver Concept
The virtual memory pool is made up with one or multiple virtual memory regions, see below figure:

• A virtual memory pool stands for the whole virtual address range that can be mapped to physical memory.
• A virtual memory region is a range of virtual address with same attributes.
• A virtual memory block is a piece of virtual address range that is dynamically mapped.
• A slot is the virtual address range between two virtual memory blocks.
• A physical memory block is a piece of physical address range that is to-be-mapped or already mapped to a virtual memory block.
• Dynamical mapping is done by calling `esp_mmap` driver API `esp_mmu_map()`. This API maps the given physical memory block to a virtual memory block which is allocated by the `esp_mmap` driver.

Relation Between Memory Blocks  When mapping a physical memory block A, block A can have one of the following relations with another previously mapped physical memory block B:

• Enclosed: block A is completely enclosed within block B, see figure below:

![Enclosed Diagram](image1)

• Identical: block A is completely the same as block B, see figure below:

![Identical Diagram](image2)

Note that `esp_mmap` driver considers the identical scenario the same as the enclosed scenario.
• Overlapped: block A is overlapped with block B, see figure below:

![Overlapped Diagram](image3)
There is a special condition, when block A entirely encloses block B, see figure below:

![Diagram showing block A enclosing block B](image)

Note that `esp_mmap` driver considers this scenario the same as the overlapped scenario.

**Driver Behaviour**

**Memory Map** You can call `esp_mmu_map()` to do a dynamical mapping. This API can allocate a certain size of virtual memory block according to the virtual memory capabilities you selected, then map this virtual memory block to the physical memory block as you requested. The `esp_mmap` driver supports mapping to one or more types of physical memory, so you should specify the physical memory target when mapping.

By default, physical memory blocks and virtual memory blocks are one-to-one mapped. This means, when calling `esp_mmu_map()`:

- If it is the enclosed scenario, this API will return an `ESP_ERR_INVALID_STATE`. The `out_ptr` will be assigned to the start virtual memory address of the previously mapped one which encloses the to-be-mapped one.
- If it is the identical scenario, this API will behaves exactly the same as the enclosed scenario.
- If it is the overlapped scenario, this API will by default return an `ESP_ERR_INVALID_ARG`. This means, `esp_mmap` driver by default does not allow mapping a physical memory address to multiple virtual memory addresses.

Specially, you can use `ESP_MMU_MMAP_FLAG_PADDR_SHARED`. This flag stands for one-to-multiple mapping between a physical address and multiple virtual addresses:

- If it is the overlapped scenario, this API will allocate a new virtual memory block as requested, then map to the given physical memory block.

**Memory Unmap** You can call `esp_mmu_unmap()` to unmap a previously mapped memory block. This API returns an `ESP_ERR_NOT_FOUND` if you are trying to unmap a virtual memory block that is not mapped to any physical memory block yet.

**Memory Address Conversion** The `esp_mmap` driver provides two helper APIs to do the conversion between virtual memory address and physical memory address:

- `esp_mmu_vaddr_to_paddr()` converts virtual address to physical address.
- `esp_mmu_paddr_to_vaddr()` converts physical address to virtual address.
Memory Synchronization  MMU supported physical memories can be accessed by one or multiple methods.

SPI flash can be accessed by SPI1 (ESP-IDF esp_flash driver APIs), or by pointers. ESP-IDF esp_flash driver APIs have already considered the memory synchronization, so users do not need to worry about this.

PSRAM can be accessed by pointers, hardware guarantees the data consistency when PSRAM is only accessed via pointers.

Thread Safety

APIs in esp_mmu_map.h are not guaranteed to be thread-safe.

API Reference

API Reference - ESP MMAP Driver

Header File

- components/esp_mm/include/esp_mmu_map.h
- This header file can be included with:

```c
#include "esp_mmu_map.h"
```

- This header file is a part of the API provided by the esp_mm component. To declare that your component depends on esp_mm, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_mm
```

or

```cmake
PRIV_REQUIRES esp_mm
```

Functions

```c
esp_err_t esp_mmu_map(esp_paddr_t paddr_start, size_t size, mmu_target_t target, mmu_mem_caps_t caps, int flags, void **out_ptr)
```

Map a physical memory block to external virtual address block, with given capabilities.

Note:  This API does not guarantee thread safety

Parameters

- **paddr_start**  — [in] Start address of the physical memory block
- **size**  — [in] Size to be mapped. Size will be rounded up by to the nearest multiple of MMU page size
- **target**  — [in] Physical memory target you’re going to map to, see mmu_target_t
- **caps**  — [in] Memory capabilities, see mmu_mem_caps_t
- **flags**  — [in] Mmap flags
- **out_ptr**  — [out] Start address of the mapped virtual memory

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG: Invalid argument, see printed logs
- ESP_ERR_NOT_SUPPORTED: Only on ESP32, PSRAM is not a supported physical memory target
- ESP_ERR_NOT_FOUND: No enough size free block to use
- ESP_ERR_NO_MEM: Out of memory, this API will allocate some heap memory for internal usage
• ESP_ERR_INVALID_STATE: Paddr is mapped already, this API will return corresponding vaddr_start of the previously mapped block. Only to-be-mapped paddr block is totally enclosed by a previously mapped block will lead to this error. (Identical scenario will behave similarly) new_block_start new_block_end

\[
\text{new_block_start new_block_end} \quad \text{Block} \quad \text{block_start block_end}
\]

\[\text{NewBlock} \quad \text{Block} \quad \text{block_start block_end}\]

**esp_err_t esp_mmu_unmap**(void *ptr)

Unmap a previously mapped virtual memory block.

**Note:** This API does not guarantee virtual thread safety

**Parameters**

ptr - [in] Start address of the virtual memory

**Returns**

• ESP_OK
• ESP_ERR_INVALID_ARG: Null pointer
• ESP_ERR_NOT_FOUND: Vaddr is not in external memory, or it is not mapped yet

**esp_err_t esp_mmu_map_get_max_consecutive_free_block_size**(mmu_mem_caps_t caps, mmu_target_t target, size_t *out_len)

Get largest consecutive free external virtual memory block size, with given capabilities and given physical target.

**Parameters**

• caps - [in] bitwise OR of MMU_MEM_CAP_* flags indicating the memory block
• target - [in] Physical memory target you’re going to map to, see mmu_target_t.
• out_len - [out] Largest free block length, in bytes.

**Returns**

• ESP_OK
• ESP_ERR_INVALID_ARG: Invalid arguments, could be null pointer

**esp_err_t esp_mmu_map_dump_mapped_blocks**(FILE *stream)

Dump all the previously mapped blocks

**Note:** This API shall not be called from an ISR.

**Note:** This API does not guarantee thread safety

**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.

**Returns**

• ESP_OK

**esp_err_t esp_mmu_vaddr_to_paddr**(void *vaddr, esp_paddr_t *out_paddr, mmu_target_t *out_target)

Convert virtual address to physical address.

**Parameters**

• vaddr - [in] Virtual address
• out_paddr - [out] Physical address
• out_target - [out] Physical memory target, see mmu_target_t

**Returns**

• ESP_OK
• ESP_ERR_INVALID_ARG: Null pointer, or vaddr is not within external memory
• ESP_ERR_NOT_FOUND: Vaddr is not mapped yet
### esp_err_t esp_mmu_paddr_to_vaddr

`esp_mmu_paddr_to_vaddr` converts a physical address to a virtual address.

**Parameters**

- `paddr` - [in] Physical address
- `target` - [in] Physical memory target, see `mmu_target_t`
- `type` - [in] Virtual address type, could be either instruction or data
- `out_vaddr` - [out] Virtual address

**Returns**

- `ESP_OK`
- `ESP_ERR_INVALID_ARG`: Null pointer
- `ESP_ERR_NOT_FOUND`: Paddr is not mapped yet

### esp_err_t esp_mmu_paddr_find_caps

`esp_mmu_paddr_find_caps` checks if the physical address is mapped, and if so, provides the capabilities of the virtual address where the physical address is mapped to.

**Parameters**

- `paddr` - [in] Physical address
- `out_caps` - [out] Bitwise OR of MMU_MEM_CAP_* flags indicating the capabilities of a virtual address where the physical address is mapped to.

**Returns**

- `ESP_OK`: Physical address successfully mapped.
- `ESP_ERR_INVALID_ARG`: Null pointer
- `ESP_ERR_NOT_FOUND`: Physical address is not mapped successfully.

#### Macros

**ESP_MMU_MMAP_FLAG_PADDR_SHARED**

- Share this mapping.
- MMU Memory Mapping Driver APIs for MMU supported memory
- Driver Backgrounds:

#### Type Definitions

```c
typedef uint32_t esp_paddr_t
```

- Physical memory type.

## 2.9.16 Memory Synchronization

### Introduction

ESP32-S2 can access its connected PSRAM via these ways:

- CPU
- DMA
By default, CPU accesses the above mentioned memory via cache. Whereas DMA accesses the memory directly, without going through cache.

This leads to potential cache data coherence issue:

- When a DMA transaction changes the content of a piece of memory, and the content has been cached already. Under this condition:
  - CPU may read stale data.
  - The stale data in the cache may be written back to the memory. The new data updated by the previous DMA transaction will be overwritten.
- CPU changes the content of an address. The content is in the cache, but not in the memory yet (cache will write back the content to the memory according to its own strategy). Under this condition:
  - The next DMA transactions to read this content from the memory will get stale data.

There are three common methods to address such cache data coherence issue:

1. Hardware based cache Coherent Interconnect, ESP32-S2 does not have such ability.
2. Use the DMA buffer from non-cacheable memory. Memory that CPU access it without going through cache is called non-cacheable memory.
3. Explicitly call a memory synchronization API to writeback the content in the cache back to the memory, or invalidate the content in the cache.

**Memory Synchronisation Driver**

The suggested way to deal with such cache data coherence issue is by using the memory synchronization API `esp_cache_msync()` provided by ESP-IDF `esp_mm` component.

**Driver Concept** Direction of the cache memory synchronization:

- `ESP_CACHE_MSYNC_FLAG_DIR_C2M`, for synchronization from cache to memory.
- `ESP_CACHE_MSYNC_FLAG_DIR_M2C`, for synchronization from memory to cache.

Type of the cache memory synchronization:

- `ESP_CACHE_MSYNC_FLAG_TYPE_DATA`, for synchronization to a data address region.
- `ESP_CACHE_MSYNC_FLAG_TYPE_INST`, for synchronization to an instruction address region.

**Driver Behaviour** Calling `esp_cache_msync()` will do a synchronization between cache and memory. The first parameter `addr` and the second parameter `size` together describe the memory region that is to be synchronized. About the third parameter `flags`:

- `ESP_CACHE_MSYNC_FLAG_DIR_C2M`. With this flag, content in the specified address region is written back to the memory. This direction is usually used after the content of an address is updated by the CPU, e.g. a memset to the address. Operation in this direction should happen before a DMA operation to the same address.
- `ESP_CACHE_MSYNC_FLAG_DIR_M2C`. With this flag, content in the specified address region is invalidated from the cache. This direction is usually used after the content of an address is updated by the DMA. Operation in this direction should happen before a CPU read operation to the same address.

The above two flags help select the synchronization direction. Specially, if neither of these two flags are used, `esp_cache_msync()` will by default select the `ESP_CACHE_MSYNC_FLAG_DIR_C2M` direction. Users are not allowed to set both of the two flags at the same time.
The above two flags help select the type of the synchronization address. Specially, if neither of these two flags are used, `esp_cache_msync()` will by default select the `ESP_CACHE_MSYNC_FLAG_TYPE_DATA` direction. Users are not allowed to set both of the two flags at the same time.

- **ESP_CACHE_MSYNC_FLAG_INVALIDATE.** This flag is used to trigger a cache invalidation to the specified address region, after the region is written back to the memory. This flag is mainly used for `ESP_CACHE_MSYNC_FLAG_DIR_C2M` direction. For `ESP_CACHE_MSYNC_FLAG_DIR_M2C` direction, behaviour is the same as if the `ESP_CACHE_MSYNC_FLAG_INVALIDATE` flag is not set.
- **ESP_CACHE_MSYNC_FLAG_UNALIGNED.** This flag force the `esp_cache_msync()` API to do synchronization without checking the address and size alignment. For more details, see chapter *Address Alignment Requirement* following.

### Address Alignment Requirement

There is address and size alignment requirement (in bytes) for using `esp_cache_msync()`. The alignment requirement comes from cache.

- An address region whose start address and size both meet the cache memory synchronization alignment requirement is defined as an **aligned address region**.
- An address region whose start address or size does not meet the cache memory synchronization alignment requirement is defined as an **unaligned address region**.

By default, if you specify an unaligned address region, `esp_cache_msync()` will return an `ESP_ERR_INVALID_ARG` error, together with the required alignment.

### Memory Allocation Helper

Cache memory synchronization is usually considered when DMA is involved. ESP-IDF provides an API to do memory allocation that can meet the alignment requirement from both the cache and the DMA.

- `esp_dma_malloc()`, this API allocates a chunk of memory that meets the alignment requirement from both the cache and the DMA.
- `esp_dma_calloc()`, this API allocates a chunk of memory that meets the alignment requirement from both the cache and the DMA. The initialized value in the memory is set to zero.

You can also use `ESP_DMA_MALLOC_FLAG_PSRAM` to allocate from the PSRAM.

### Warning for Address Alignment Requirement

You can set the `ESP_CACHE_MSYNC_FLAG_UNALIGNED` flag to bypass such check. Note you should be very careful about using this flag. Cache memory synchronization to an unaligned address region may silently corrupt the memory.

For example, assume:

- alignment requirement is 0x40 bytes.
- a call to `esp_cache_msync()`, with `ESP_CACHE_MSYNC_FLAG_DIR_M2C` | `ESP_CACHE_MSYNC_FLAG_UNALIGNED` flags, the specified address region is 0x4000_0020 ~ 0x4000_0060 (see data C in below graph).

Above settings will trigger a cache invalidation to the address region 0x4000_0000 ~ 0x4000_0080, see `sync item0` and `sync item1` in the below graph.

If the content in 0x4000_0000 ~ 0x4000_0020 (data A in the below graph) or 0x4000_0060 ~ 0x4000_0080 (data B in the below graph) are not written back to the memory yet, then these data A and data B will be discarded.
Chapter 2. API Reference

API Reference

API Reference - ESP Msync Driver

Header File

- components/esp_mm/include/esp_cache.h
- This header file can be included with:
  
  ```
  #include "esp_cache.h"
  ```

- This header file is a part of the API provided by the esp_mm component. To declare that your component depends on esp_mm, add the following to your CMakeLists.txt:

  ```
  REQUIRES esp_mm
  ```

  or

  ```
  PRIV_REQUIRES esp_mm
  ```

Functions

`esp_err_t esp_cache_msync (void *addr, size_t size, int flags)`

Memory sync between Cache and storage memory.

For cache-to-memory (C2M) direction:

- For cache writeback supported chips (you can refer to SOC_CACHE_WRITEBACK_SUPPORTED in soc_caps.h)
  
  - This API will do a writeback to synchronise between cache and storage memory
With ESP_CACHE_MSYNC_FLAG_INVALIDATE, this API will also invalidate the values that just written

Note: although ESP32 is with PSRAM, but cache writeback isn’t supported, so this API will do nothing on ESP32

• For other chips, this API will do nothing. The out-of-sync should be already dealt by the SDK

For memory-to-cache (M2C) direction:

• This API will by default do an invalidation

This API is cache-safe and thread-safe

Note: If you don’t set direction (ESP_CACHE_MSYNC_FLAG_DIR_x flags), this API is by default C2M direction

Note: If you don’t set type (ESP_CACHE_MSYNC_FLAG_TYPE_x flags), this API is by default doing msync for data

Note: You should not call this during any Flash operations (e.g. esp_flash APIs, nvs and some other APIs that are based on esp_flash APIs)

Note: If XIP_From_PSRAM is enabled (by enabling both CONFIG_SPIRAM_FETCH_INSTRUCTIONS and CONFIG_SPIRAM_RODATA), you can call this API during Flash operations

Parameters

• addr [in] Starting address to do the msync
• size [in] Size to do the msync
• flags [in] Flags, see ESP_CACHE_MSYNC_FLAG_x

Returns

• ESP_OK:
  – Successful msync
  – For C2M direction, if this chip doesn’t support cache writeback, if the input addr is a cache supported one, this API will return ESP_OK
• ESP_ERR_INVALID_ARG: Invalid argument, not cache supported addr, see printed logs

Macros

ESP_CACHE_MSYNC_FLAG_INVALIDATE
Do an invalidation.
Cache msync flags

• For cache-to-memory (C2M) direction: setting this flag will start an invalidation after the cache writeback operation
• For memory-to-cache (M2C) direction: setting / unsetting this flag will behave similarly, trigger an invalidation

ESP_CACHE_MSYNC_FLAG_UNALIGNED
Allow msync to a address block that are not aligned to the data cache line size.

ESP_CACHE_MSYNC_FLAG_DIR_C2M
Cache msync direction: from Cache to memory.
Note: If you don’t set direction (ESP_CACHE_MSYNC_FLAG_DIR_x flags), it is by default cache-to-memory (C2M) direction.

ESP_CACHE_MSYNC_FLAG_DIR_M2C
Cache msync direction: from memory to Cache.

ESP_CACHE_MSYNC_FLAG_TYPE_DATA
Cache msync type: data.

Note: If you don’t set type (ESP_CACHE_MSYNC_FLAG_TYPE_x flags), it is by default data type.

ESP_CACHE_MSYNC_FLAG_TYPE_INST
Cache msync type: instruction.

API Reference - ESP DMA Utils

Header File
- components/esp_hw_support/include/esp_dma_utils.h
- This header file can be included with:
  ```
  #include "esp_dma_utils.h"
  ```

Functions

**esp_err_t esp_dma_malloc** (size_t size, uint32_t flags, void **out_ptr, size_t *actual_size)
Helper function for malloc a DMA capable memory buffer.

Parameters
- **size** -[in] Size in bytes, the amount of memory to allocate
- **flags** -[in] Flags, see ESP_DMA_MALLOC_FLAG_x
- **out_ptr** -[out] A pointer to the memory allocated successfully
- **actual_size** -[out] Actual size for allocation in bytes, when the size you specified doesn’t meet the DMA alignment requirements, this value might be bigger than the size you specified. Set null if you don’t care this value.

Returns
- ESP_OK:
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_NO_MEM: No enough memory for allocation

**esp_err_t esp_dma calloc** (size_t n, size_t size, uint32_t flags, void **out_ptr, size_t *actual_size)
Helper function for calloc a DMA capable memory buffer.

Parameters
- **n** -[in] Number of continuing chunks of memory to allocate
- **size** -[in] Size of one chunk, in bytes
- **flags** -[in] Flags, see ESP_DMA_MALLOC_FLAG_x
- **out_ptr** -[out] A pointer to the memory allocated successfully
- **actual_size** -[out] Actual size for allocation in bytes, when the size you specified doesn’t meet the cache alignment requirements, this value might be bigger than the size you specified. Set null if you don’t care this value.

Returns
- ESP_OK:
- ESP_ERR_INVALID_ARG: Invalid argument
Chapter 2. API Reference

- ESP_ERR_NO_MEM: No enough memory for allocation

bool esp_dma_is_buffer_aligned (const void *ptr, size_t size, esp_dma_buf_location_t location)
Helper function to check if a buffer meets DMA alignment requirements.

Parameters
- ptr [in] Pointer to the buffer
- size [in] Size of the buffer
- location [in] Location of the DMA buffer, see esp_dma_buf_location_t

Returns
- True: Buffer is aligned
- False: Buffer is not aligned, or buffer is not DMA capable

Macros

ESP_DMA_MALLOC_FLAG_PSRAM
Memory is in PSRAM.
DMA malloc flags

Enumerations

enum esp_dma_buf_location_t
DMA buffer location.
Values:

enumerator ESP_DMA_BUF_LOCATION_INTERNAL
DMA buffer is in internal memory.

enumerator ESP_DMA_BUF_LOCATION_PSRAM
DMA buffer is in PSRAM.

2.9.17 Heap Memory Debugging

Overview

ESP-IDF integrates tools for requesting heap information, heap corruption detection, and heap tracing. These can help track down memory-related bugs.
For general information about the heap memory allocator, see Heap Memory Allocation.

Heap Information

To obtain information about the state of the heap, call the following functions:

- heap_caps_get_free_size() can 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, which is also the largest single allocation currently possible. Tracking this value and comparing it to the total free heap allows you to detect heap fragmentation.
- heap_caps_get_minimum_free_size() can be used to track the heap “low watermark” 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 of the information returned by `heap_caps_get_info()` to stdout.
• `heap_caps_dump()` and `heap_caps_dump_all()` output detailed information about the structure of each block in the heap. Note that this can be a large amount of output.

Heap Allocation and Free Function Hooks

Heap allocation and free detection hooks allow you to be notified of every successful allocation and free operation:

- Providing a definition of `esp_heap_trace_alloc_hook()` allows you to be notified of every successful memory allocation operation
- Providing a definition of `esp_heap_trace_free_hook()` allows you to be notified of every successful memory-free operations

This feature can be enabled by setting the `CONFIG_HEAP_USE_HOOKS` option. `esp_heap_trace_alloc_hook()` and `esp_heap_trace_free_hook()` have weak declarations (e.g., `__attribute__((weak))`), thus it is not necessary to provide declarations for both hooks. Given that it is technically possible to allocate and free memory from an ISR (though strongly discouraged from doing so), the `esp_heap_trace_alloc_hook()` and `esp_heap_trace_free_hook()` can potentially be called from an ISR.

It is not recommended to perform (or call API functions to perform) blocking operations or memory allocation/free operations in the hook functions. In general, the best practice is to keep the implementation concise and leave the heavy computation outside of the hook functions.

The example below shows how to define the allocation and free function hooks:

```c
#include "esp_heap_caps.h"

void esp_heap_trace_alloc_hook(void* ptr, size_t size, uint32_t caps)
{
    ...
}

void esp_heap_trace_free_hook(void* ptr)
{
    ...
}

void app_main()
{
    ...
}
```

Heap Corruption Detection

Heap corruption detection allows you to detect various types of heap memory errors:

- Out-of-bound writes & buffer overflows
- Writes to freed memory
- Reads from freed or uninitialized memory

**Assertions** The heap implementation (heap-multi_heap.c, etc.) includes numerous assertions that will fail if the heap memory is corrupted. To detect heap corruption most effectively, ensure that assertions are enabled in the project configuration via the `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL` option.

If a heap integrity assertion fails, a line will be printed like `CORRUPT HEAP: multi_heap.c:225 detected at 0x3ffbb71c`. The memory address printed is the address of the heap structure that has corrupt content.

It is also possible to manually check heap integrity by calling `heap_caps_check_integrity_all()` or related functions. This function checks all of the requested heap memory for integrity and can be used even if assertions
are disabled. If the integrity checks detects an error, it will print the error along with the address(es) of corrupt heap structures.

**Memory Allocation Failed Hook** Users can use `heap_caps_register_failed_alloc_callback()` to register a callback that is invoked every time an allocation operation fails.

Additionally, users can enable the `CONFIG_HEAP_ABORT_WHEN_ALLOCATION_FAILS`, which will automatically trigger a system abort if any allocation operation fails.

The example below shows how to register an allocation failure callback:

```c
#include "esp_heap_caps.h"

void heap_caps_alloc_failed_hook(size_t requested_size, uint32_t caps, const char *function_name)
{
    printf("%s was called but failed to allocate %d bytes with 0x%X capabilities. \n", function_name, requested_size, caps);
}

void app_main()
{
    ... 
    esp_err_t error = heap_caps_register_failed_alloc_callback(heap_caps_alloc_failed_hook);
    ...
    void *ptr = heap_caps_malloc(allocation_size, MALLOC_CAP_DEFAULT);
    ...
}
```

**Finding Heap Corruption** Memory corruption can be one of the hardest classes of bugs to find and fix, as the source of the corruption could be completely unrelated to the symptoms of the corruption. Here are some tips:

- A crash with a CORRUPT HEAP: message usually includes a stack trace, but this stack trace is rarely useful. The crash is the symptom of memory corruption when the system realizes the heap is corrupt. But usually, the corruption happens elsewhere and earlier in time.
- Increasing the heap memory debugging Configuration level to “Light impact” or “Comprehensive” gives 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 helps 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 that has been 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 do not have JTAG, but you do know roughly when the corruption happens, set a watchpoint in software just beforehand via `esp_cpu_set_watchpoint()`. A fatal exception will occur when the watchpoint triggers. The following is an example of how to use the function - `esp_cpu_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 do not know which CPU is corrupting memory, call this function on both CPUs.
- For buffer overflows, heap tracing in HEAP_TRACE_ALL mode tells which callers are allocating which addresses from the heap. See Heap Tracing To Find Heap Corruption for more details. You can try to find the function that allocates memory with an address immediately before the corrupted address, since it is probably the function that 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 or 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 option `CONFIG_HEAP_CORRUPTION_DETECTION` can be set to one of the following three levels:

**Basic (No Poisoning)** This is the default level. By default, no special heap corruption features are enabled, but the provided assertions are enabled. 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 checks 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 that 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).

With basic heap corruption checks, most out-of-bound writes can be detected and the number of overrun bytes before a failure is detected depends on the properties of the heap. However, the Light Impact mode is more precise as even a single-byte overrun can be detected.

Enabling light-impact checking increases the memory usage. Each individual allocation uses 9 to 12 additional bytes of memory depending on alignment.

Each time `heap_caps_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 functions involve checking that the first 4 bytes of an allocated block (before the buffer is returned to the user) should be the word `0xABBA1234`, and the last 4 bytes of the allocated block (after the buffer is returned to the user) should be the word `0xBAAD5678`.

Different values usually indicate buffer underrun or overrun. Overrun indicates that when writing to memory, the data written exceeds the size of the allocated memory, resulting in writing to an unallocated memory area; underrun indicates that when reading memory, the data read exceeds the allocated memory and reads data from an unallocated memory area.

**Comprehensive** This level incorporates the “light impact” detection features plus additional checks for uninitialized-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 mode has a substantial impact on runtime performance, as all memory needs to be set to the allocation patterns each time a `heap_caps_malloc()` or `heap_caps_free()` completes, and the memory also needs to be checked each time. However, this mode 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 when reading or writing an address related to `0xCE-CECECE` in Comprehensive mode, it indicates that it has read uninitialized memory. The application should be changed to either use `heap_caps_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 ESP-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 `0xFEFE-FEFE`, this indicates that it is reading heap memory after it has been freed, i.e., a “use-after-free bug”. The application should be changed to not access heap memory after it has been freed.
If a call to `heap_caps_malloc()` or `heap_caps_realloc()` causes a crash because it was expected to find the pattern `0xFEFEFEFE` in free memory and a different pattern was found, it indicates that the app has a use-after-free bug where it is writing to memory that 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 expected to find a given pattern, and will error out if 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 behavior is the same as for the 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 or underrun.

**Heap Task Tracking**

Heap Task Tracking can be used to get per-task info for heap memory allocation. The application has to specify the heap capabilities for which the heap allocation is to be tracked.

Example code is provided in `system/heap_task_tracking`.

**Heap Tracing**

Heap Tracing allows the tracing of code which allocates or frees memory. Two tracing modes are supported:

- Standalone. In this mode, traced data are kept on-board, so the size of the gathered information is limited by the buffer assigned for that purpose, and the 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 traced data are sent to the host over JTAG connection using app_trace library. Later on, they can be analyzed using special tools.

Heap tracing can perform two functions:

- Leak checking: find memory that is allocated and never freed.
- Heap use analysis: show all functions that are allocating or 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 `heap_caps_get_free_size()` or related functions in `heap information` 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 have identified the code which you think is leaking:

- Enable the `CONFIG_HEAP_TRACING_DEST` option.
- Call the function `heap_trace_init_standalone()` early in the program, to register a buffer that can be used to record the memory trace.
- Call the function `heap_trace_start()` to begin recording all mallocs or 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.

The following code snippet demonstrates how application code would typically initialize, start, and stop heap tracing:

```c
#include "esp_heap_trace.h"

#define NUM_RECORDS 100
static heap_trace_record_t trace_record[NUM_RECORDS]; // This buffer must be in...
```
... 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 has a similar format to the following example:

```
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
```

**Note:** The above example output uses IDF Monitor to automatically decode PC addresses to their source files and line numbers.

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 that has not already been freed, a line is printed with:

- XX bytes is the number of bytes allocated.
- @ 0x... is the heap address returned from heap_caps_malloc() or heap_caps_calloc()
- Internal or PSRAM is the general location of the allocated memory.
- CPU x is the CPU (0 or 1) running when the allocation was made.
- ccount 0x... is the CCOUNT (CPU cycle count) register value the allocation was made. The value is different for CPU 0 vs CPU 1.
- caller 0x... gives the call stack of the call to heap_caps_malloc() or heap_caps_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 > `CONFIG_HEAP_TRACING_STACK_DEPTH`. Up to 32 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 the ‘leaked’ bytes (bytes allocated but not freed while the trace is running) is printed together with the total number of allocations it represents.

A warning will be printed if the trace buffer was not large enough to hold all the allocations 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 have 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_DESTINATION1` and select Trace memory.
- In the project configuration menu, navigate to Component settings > Application Level Tracing > FreeRTOS SystemView Tracing and enable `CONFIG_APPTRACE_SV_ENABLE`.
- Call the function `heap_trace_init_tohost()` early in the program, to initialize the JTAG heap tracing module.
- Call the function `heap_trace_start()` to begin recording all memory allocation and free calls in the system. Call this immediately before the piece of code which you suspect is leaking memory. In host-based mode, the argument to this function is ignored, and the heap tracing module behaves like `HEAP_TRACE_ALL` is passed, i.e., 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.

The following code snippet demonstrates how application code would typically initialize, start, and stop host-based mode heap tracing:

```c
#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 analyze heap trace, do the following on the host:

1. Build the program and download it to the target as described in *Step 5. First Steps on ESP-IDF*.
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 a special `gdbinit` file:
target remote :3333
mon reset halt
maintenance flush register cache
tb heap_trace_start
commands
mon esp sysview start file://tmp/heap.svdat
c
end
tb heap_trace_stop
commands
mon esp sysview stop
c

Using this file GDB can connect to the target, reset it, and start tracing when the program hits breakpoint at `heap_trace_start()`. Tracing will be stopped when the program hits breakpoint at `heap_trace_stop()`. Traced data will be saved to `/tmp/heap_log.svdat`.

4. Run GDB using `xtensa-esp32s2-elf-gdb -x gdbinit </path/to/program/elf>`.
5. Quit GDB when the program stops at `heap_trace_stop()`. Traced 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 has a similar format to the following example:

```
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 @ 0x3ffaffd8 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 @ 0x3ffaffe0 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 @ 0x3ffaffe0 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: Allocated 2 bytes @ 0x3ffaffe8 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)
[0.002798700] HEAP: Allocated 4 bytes @ 0x3ffaffe8 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)
(continues on next page)
```
Heap Tracing To Find Heap Corruption

Heap tracing can also be used to help track down heap corruption. When a region in the heap is corrupted, it may be from some other part of the program that allocated memory at a nearby address.

If you have an approximate idea of when the corruption occurred, enabling heap tracing in `HEAP_TRACE_ALL` mode allows you to record all the memory allocation functions used and the corresponding allocation addresses.

Using heap tracing in this way is very similar to memory leak detection as described above. For memories that are allocated and not freed, the output is the same. However, records will also be shown for memory that 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 the performance of heap allocation or free operations even when heap tracing is not running.

When heap tracing is running, heap allocation or free operations are substantially slower than when heap tracing is stopped. Increasing the depth of stack frames recorded for each allocation (see above) also increases this performance impact.

To mitigate the performance loss when the heap tracing is enabled and active, enable `CONFIG_HEAP_TRACE_HASH_MAP`. With this configuration enabled, a hash map mechanism will be used to handle the heap trace records, thus considerably decreasing the heap allocation or free execution time. The size of the hash map can be modified by setting the value of `CONFIG_HEAP_TRACE_HASH_MAP_SIZE`.

By default, the hash map is placed into internal RAM. It can also be placed into external RAM if `CONFIG_HEAP_TRACE_HASH_MAP_IN_EXT_RAM` is enabled. In order to enable this configuration, make sure to enable `CONFIG_SPIRAM` and `CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY`.

False-Positive Memory Leaks

Not everything printed by `heap_trace_dump()` is necessarily a memory leak. The following cases may also be printed:

- Any memory that is allocated after `heap_trace_start()` but freed after `heap_trace_stop()` appears in the leaked dump.
- Allocations may be made by other tasks in the system. Depending on the timing of these tasks, it is quite possible that this memory is freed after `heap_trace_stop()` is called.
- The first time a task uses stdout - e.g., when it calls `heap_caps_printf()` - a lock, i.e., RTOS mutex semaphore, is allocated by the libc. This allocation lasts until the task is deleted.
- Certain uses of `heap_caps_printf()`, such as printing floating point numbers and allocating some memory from the heap on demand. These allocations last until the task is deleted.
- The Bluetooth, Wi-Fi, and TCP/IP libraries 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 has been received or transmitted by the lower levels of the network during the heap tracing.
- TCP connections retain some memory even after they are closed due to the `TIME_WAIT` state. Once the `TIME_WAIT` period is 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
• components/heap/include/esp_heap_trace.h
• This header file can be included with:

```
#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);`

**Parameters**
- `record_buffer` – Provide a buffer to use for heap trace data. Note: External RAM is allowed, but it prevents recording allocations made from ISR’s.
- `num_records` – Size of the heap trace buffer, as number of record structures.

**Returns**
- 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.

### `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.

**Returns**
- 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.

**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.)

**Returns**
- 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.

### `esp_err_t heap_trace_stop(void)`
Stop heap tracing.

**Returns**
- ESP_ERR_NOT_SUPPORTED Project was compiled without heap tracing enabled in menuconfig.
- ESP_ERR_INVALID_STATE Heap tracing was not in progress.
Chapter 2. API Reference

- 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).

**Returns**
- 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.

**Parameters**
- **index** – Index (zero-based) of the record to return.
- **record** – [out] Record where the heap trace record will be copied.

**Returns**
- 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.

**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.

**void heap_trace_dump_caps (const uint32_t caps)**
Dump heap trace from the memory of the capabilities passed as parameter.

**Parameters**
- **caps** – Capability(ies) of the memory from which to dump the trace. Set MAL-LOC_CAP_INTERNAL to dump heap trace data from internal memory. Set MAL-LOC_CAP_SPIRAM to dump heap trace data from PSRAM. Set both to dump both heap trace data.

**esp_err_t heap_trace_summary (heap_trace_summary_t *summary)**
Get summary information about the result of a heap trace.

**Note:** It is safe to call this function while heap tracing is running.
Chapter 2. API Reference

Structures

struct `heap_trace_record_t`

- **Tracerecord datatype.** 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. If NULL, then this record is empty.
  - `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.)

struct `heap_trace_summary_t`

- **Stores information about the result of a heap trace.**

  **Public Members**

  - `heap_trace_mode_t mode`
    - The heap trace mode we just completed / are running.
  - `size_t total_allocations`
    - The total number of allocations made during tracing.
  - `size_t total_frees`
    - The total number of frees made during tracing.
  - `size_t count`
    - The number of records in the internal buffer.
  - `size_t capacity`
    - The capacity of the internal buffer.
  - `size_t high_water_mark`
    - The maximum value that `count` got to.
  - `size_t has_overflowed`
    - True if the internal buffer overflowed at some point.
Macros

**CONFIG_HEAP_TRACING_STACK_DEPTH**

Type Definitions

typedef struct heap_trace_record { \n  \n} heap_trace_record_t

Trace record data type. Stores information about an allocated region of memory.

Enumerations

enum heap_trace_mode_t

Values:

- enumerator HEAP_TRACE_ALL
- enumerator HEAP_TRACE_LEAKS

2.9.18 High Resolution Timer (ESP Timer)

Overview

Although FreeRTOS provides software timers, FreeRTOS software timers have a few limitations:

- Maximum resolution is equal to the RTOS tick period
- Timer callbacks are dispatched from a low-priority timer service (i.e., daemon) task. This task can be preempted by other tasks, leading to decreased precision and accuracy.

Although hardware timers are not subject to the limitations mentioned, they may not be as user-friendly. For instance, application components may require timer events to be triggered at specific future times, but hardware timers typically have only one “compare” value for interrupt generation. This necessitates the creation of an additional system on top of the hardware timer to keep track of pending events and ensure that callbacks are executed when the corresponding hardware interrupts occur.

*esp_timer* set of APIs provides one-shot and periodic timers, microsecond time resolution, and 64-bit range.

Internally, *esp_timer* uses a 64-bit hardware timer. The exact hardware timer implementation used depends on the target, where SYSTIMER is used for ESP32-S2.

Timer callbacks can be dispatched by two methods:

- **ESP_TIMER_TASK**. 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.

- **ESP_TIMER_ISR**. Available only if `CONFIG_ESP_TIMER_SUPPORTS_ISR_DISPATCH_METHOD` is enabled (by default disabled).

If other tasks with a priority higher than esp_timer are running, callback dispatching will be delayed until the esp_timer task has a chance to run. For example, this will happen if an SPI Flash operation is in progress.

*ESP_TIMER_ISR*. Timer callbacks are dispatched directly from the timer interrupt handler. This method is useful for some simple callbacks which aim for lower latency.
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 \texttt{esp_timer}. If \texttt{esp_timer_start_once()} is called with a timeout value of less than 20 us, the callback will be dispatched only after approximately 20 us.

Periodic \texttt{esp_timer} also imposes a 50 us restriction on the minimal timer period. Periodic software timers with a period of less than 50 us 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 a small period is required.

**Using \texttt{esp_timer} APIs**

A single timer is represented by \texttt{esp_timer_handle_t} type. Each timer has a callback function associated with it. This callback function is called from the \texttt{esp_timer} task each time the timer elapses.

- To create a timer, call \texttt{esp_timer_create()}.
- To delete the timer when it is no longer needed, call \texttt{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 \texttt{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 \texttt{esp_timer_start_periodic()}, passing the period with which the callback should be called. The timer keeps running until \texttt{esp_timer_stop()} is called.

Note that the timer must not be running when \texttt{esp_timer_start_once()} or \texttt{esp_timer_start_periodic()} is called. To restart a running timer, call \texttt{esp_timer_stop()} first, then call one of the start functions.

**Callback Functions**

**Note:** Keep the callback functions as short as possible. Otherwise, it will affect all timers.

Timer callbacks that are processed by the ESP_TIMER_ISR method should not call the context switch call - \texttt{portYIELD_FROM_ISR()}. Instead, use the \texttt{esp_timer_isr_dispatch_need_yield()} function. The context switch will be done after all ISR dispatch timers have been processed if required by the system.

**\texttt{esp_timer} During Light-sleep**

During Light-sleep, the \texttt{esp_timer} counter stops and no callback functions are called. Instead, the time is counted by the RTC counter. Upon waking up, the system gets the difference between the counters and calls a function that advances the \texttt{esp_timer} counter. Since the counter has been advanced, the system starts calling callbacks that were not called during sleep. The number of callbacks depends on the duration of the sleep and the period of the timers. It can lead to the overflow of some queues. This only applies to periodic timers, since one-shot timers are only called once.

This behavior can be changed by calling \texttt{esp_timer_stop()} before sleeping. In some cases, this can be inconvenient, and instead of the stop function, you can use the \texttt{skip_unhandled_events} option during \texttt{esp_timer_create()}. When the \texttt{skip_unhandled_events} is true, if a periodic timer expires one or more times during Light-sleep, then only one callback is called on wake.

Using the \texttt{skip_unhandled_events} option with automatic Light-sleep (see Power Management APIs) helps to reduce the power consumption of the system when it is in Light-sleep. The duration of Light-sleep is also in part determined by the next event occurs. Timers with \texttt{skip_unhandled_events} option does not wake up the system.
Handling Callbacks

`esp_timer` is designed to achieve a high-resolution and low-latency timer with the ability to handle delayed events. If the timer is late, then the callback will be called as soon as possible, and it will not be lost. In the worst case, when the timer has not been processed for more than one period (for periodic timers), the callbacks will be called one after the other without waiting for the set period. This can be bad for some applications, and the `skip_unhandled_events` option is introduced to eliminate this behavior. If `skip_unhandled_events` is set, then a periodic timer that has expired multiple times without being able to call the callback will still result in only one callback event once processing is possible.

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 the usage of `esp_timer` APIs: `system/esp_timer`.

API Reference

Header File

- `components/esp_timer/include/esp_timer.h`
- This header file can be included with:

```c
#include "esp_timer.h"
```

- This header file is a part of the API provided by the `esp_timer` component. To declare that your component depends on `esp_timer`, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_timer
```

or

```cmake
PRIV_REQUIRES esp_timer
```

Functions

`esp_err_t esp_timer_early_init(void)`

Minimal initialization of `esp_timer`.

This function can be called very early in startup process, after this call only `esp_timer_get_time` function can be used.

**Note:** This function is called from startup code. Applications do not need to call this function before using other `esp_timer` APIs.

**Returns**

- `ESP_OK` on success


**esp_err_t esp_timer_init** (void)

Initialize esp_timer library.

This function will be called from startup code on every core if CONFIG_ESP_TIMER_ISR_AFFINITY_NO_AFFINITY is enabled. It allocates the timer ISR on MULTIPLE cores and creates the timer task which can be run on any core.

**Note:** This function is called from startup code. Applications do not need to call this function before using other esp_timer APIs. Before calling this function, esp_timer_early_init must be called by the startup code.

**Returns**
- 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

**Returns**
- 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.

**Parameters**
- **create_args** – Pointer to a structure with timer creation arguments. Not saved by the library, can be allocated on the stack.
- **out_handle** – [out] Output, pointer to esp_timer_handle_t variable which will hold the created timer handle.

**Returns**
- 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

**esp_err_t esp_timer_start_once** (esp_timer_handle_t timer, uint64_t timeout_us)

Start one-shot timer.

**Parameters**
- **timer** – Timer handle created using esp_timer_create
- **timeout_us** – Timer timeout, in microseconds relative to the current moment

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is already running
**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.

**Parameters**
- `timer` - timer handle created using esp_timer_create
- `period` - timer period, in microseconds

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is already running

**esp_err_t esp_timer_restart (esp_timer_handle_t timer, uint64_t timeout_us)**

Restart a currently running timer.

If the given timer is a one-shot timer, the timer is restarted immediately and will timeout once in `timeout_us` microseconds. If the given timer is a periodic timer, the timer is restarted immediately with a new period of `timeout_us` microseconds.

**Parameters**
- `timer` - timer Handle created using esp_timer_create
- `timeout_us` - Timeout, in microseconds relative to the current time. In case of a periodic timer, also represents the new period.

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the handle is invalid
- ESP_ERR_INVALID_STATE if the timer is not running

**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.

**Parameters**
- `timer` - timer handle created using esp_timer_create

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_STATE if the timer is not running

**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.

**Parameters**
- `timer` - timer handle allocated using esp_timer_create

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_STATE if the timer is running

**int64_t esp_timer_get_time (void)**

Get time in microseconds since boot.

**Returns**
- number of microseconds since underlying timer has been started

**int64_t esp_timer_get_next_alarm (void)**

Get the timestamp when the next timeout is expected to occur.

**Returns**
- Timestamp of the nearest timer event, in microseconds. The timebase is the same as for the values returned by esp_timer_get_time.

**int64_t esp_timer_get_next_alarm_for_wake_up (void)**

Get the timestamp when the next timeout is expected to occur skipping those which have skip_unhandled_events flag.
Returns  Timestamp of the nearest timer event, in microseconds. The timebase is the same as for the values returned by esp_timer_get_time.

```c
esp_err_t esp_timer_get_period(esp_timer_handle_t timer, uint64_t *period)
```

Get the period of a timer.

This function fetches the timeout period of a timer.

**Note:** The timeout period is the time interval with which a timer restarts after expiry. For one-shot timers, the period is 0 as there is no periodicity associated with such timers.

**Parameters**
- `timer` — timer handle allocated using esp_timer_create
- `period` — memory to store the timer period value in microseconds

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the arguments are invalid

```c
esp_err_t esp_timer_get_expiry_time(esp_timer_handle_t timer, uint64_t *expiry)
```

Get the expiry time of a one-shot timer.

This function fetches the expiry time of a one-shot timer.

**Note:** This API returns a valid expiry time only for a one-shot timer. It returns an error if the timer handle passed to the function is for a periodic timer.

**Parameters**
- `timer` — timer handle allocated using esp_timer_create
- `expiry` — memory to store the timeout value in microseconds

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if the arguments are invalid
- ESP_ERR_NOT_SUPPORTED if the timer type is periodic

```c
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

**Parameters**
- `stream` — stream (such as stdout) to dump the information to

**Returns**
- ESP_OK on success
- ESP_ERR_NO_MEM if can not allocate temporary buffer for the output
void esp_timer_isr_dispatch_need_yield (void)
Requests a context switch from a timer callback function.

This only works for a timer that has an ISR dispatch method. The context switch will be called after all ISR
dispatch timers have been processed.

bool esp_timer_is_active (esp_timer_handle_t timer)
Returns status of a timer, active or not.

This function is used to identify if the timer is still active or not.

Parameters
timer — timer handle created using esp_timer_create

Returns
• 1 if timer is still active
• 0 if timer is not active.

esp_err_t esp_timer_new_etm_alarm_event (esp_etm_event_handle_t *out_event)
Get the ETM event handle of esp_timer underlying alarm event.

Note: The created ETM event object can be deleted later by calling esp_etm_del_event

esp_err_t esp_timer_new_etm_alarm_event (esp_etm_event_handle_t *out_event)
Get the ETM event handle of esp_timer underlying alarm event.

Note: The ETM event is generated by the underlying hardware &##8212; systimer, therefore, if the esp_timer
is not clocked by systimer, then no ETM event will be generated.

Parameters
out_event — [out] Returned ETM event handle

Returns
• ESP_OK Success
• ESP_ERR_INVALID_ARG Parameter error

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.

disp"
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.

  Param arg  pointer to opaque user-specific data

Enumerations

enum esp_timer_dispatch_t
  Method for dispatching timer callback.

  Values:

  enumerator ESP_TIMER_TASK
    Callback is called from timer task.

  enumerator ESP_TIMER_MAX
    Count of the methods for dispatching timer callback.

2.9.19 Internal and Unstable APIs

This section is listing some APIs that are internal or likely to be changed or removed in the next releases of ESP-IDF.

API Reference

Header File

  • components/esp_rom/include/esp_rom_sys.h
  • This header file can be included with:

     #include "esp_rom_sys.h"

Functions

void esp_rom_software_reset_system (void)
  Software Reset digital core include RTC.

  It is not recommended to use this function in esp-idf, use esp_restart() instead.

void esp_rom_software_reset_cpu (int cpu_no)
  Software Reset cpu core.

  It is not recommended to use this function in esp-idf, use esp_restart() instead.

  Parameters cpu_no  ⚶: The CPU to reset, 0 for PRO CPU, 1 for APP CPU.

int esp_rom_printf (const char *fmt, ...)
  Print formatted string to console device.

  Note:  float and long long data are not supported!

  Parameters
     • fmt  – Format string
• ...–Additional arguments, depending on the format string

**Returns** int: Total number of characters written on success; A negative number on failure.

```c
void esp_rom_delay_us (uint32_t us)
```

Pauses execution for us microseconds.

**Parameters**

- `us` – Number of microseconds to pause

```c
void esp_rom_install_channel_putchar (int channel, void (*putc)(char c))
```

`esp_rom_putchar` can print message to different channels simultaneously. This function can help install the low level `putc` function for `esp_rom_printf`.

**Parameters**

- `channel` – Channel number (starting from 1)
- `putc` – Function pointer to the `putc` implementation. Set NULL can disconnect `esp_rom_putchar` with `putc`.

```c
void esp_rom_install_uart_putchar (void)
```

Install UART1 as the default console channel, equivalent to `esp_rom_install_channel_putchar(1, esp_rom_uart_putchar)`

```c
soc_reset_reason_t esp_rom_get_reset_reason (int cpu_no)
```

Get reset reason of CPU.

**Parameters**

- `cpu_no` – CPU number

**Returns** Reset reason code (see in soc/reset_reasons.h)

```c
void esp_rom_install_uart_putchar (int cpu_core, uint32_t periph_intr_id, uint32_t cpu_intr_num)
```

Route peripheral interrupt sources to CPU’s interrupt port by matrix.

Usually there’re 4 steps to use an interrupt:

a. Route peripheral interrupt source to CPU. e.g. `esp_rom_route_intr_matrix(0, ETS_WIFI_MAC_INTR_SOURCE, ETS_WMAC_INTR_NUM)`

b. Set interrupt handler for CPU

c. Enable CPU interrupt

d. Enable peripheral interrupt

**Parameters**

- `cpu_core` – The CPU number, which the peripheral interrupt will inform to
- `periph_intr_id` – The peripheral interrupt source number
- `cpu_intr_num` – The CPU (external) interrupt number. On targets that use CLIC as their interrupt controller, this number represents the external interrupt number. For example, passing `cpu_intr_num = i` to this function would in fact bind peripheral source to CPU interrupt `CLIC_EXT_INTR_NUM_OFFSET + i`.

```c
uint32_t esp_rom_get_cpuTicks_per_us (void)
```

Get the real CPU ticks per us.

**Returns** CPU ticks per us

```c
void esp_rom_set_cpuTicks_per_us (uint32_t ticks_per_us)
```

Set the real CPU tick rate.

---

**Note:** Call this function when CPU frequency is changed, otherwise the `esp_rom_delay_us` can be inaccurate.

**Parameters**

- `ticks_per_us` – CPU ticks per us
2.9.20 Interrupt Allocation

Overview

The ESP32-S2 has one core, with 32 interrupts. Each interrupt has a fixed priority, most (but not all) interrupts are connected to the interrupt matrix.

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_intrstatus()`). It can use the flags passed to this function to specify the type, priority, and trigger method of the interrupt to allocate. The interrupt allocation code will then find an applicable interrupt, use the interrupt matrix to hook it up to the peripheral, and install the given interrupt handler and ISR to it.

The interrupt allocator presents two different types of interrupts, namely shared interrupts and non-shared interrupts, both of which require different handling. Non-shared interrupts will allocate a separate interrupt for every `esp_intr_alloc()` call, and this interrupt is use solely for the peripheral attached to it, with only one ISR that will get called. Shared interrupts can have multiple peripherals triggering them, 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 check if any action is required.

Non-shared interrupts can be either level- or edge-triggered. Shared interrupts can only be level interrupts due to the chance of missed interrupts when edge interrupts are used.

To illustrate why shared interrupts can only be level-triggered, take the scenario where peripheral A and peripheral B share the same edge-triggered interrupt. Peripheral B triggers an interrupt and sets its interrupt signal high, causing a low-to-high edge, which in turn latches the CPU’s interrupt bit and triggers the ISR. The ISR executes, checks that peripheral A did not trigger an interrupt, and proceeds to handle and clear peripheral B’s interrupt signal. Before the ISR returns, the CPU clears its interrupt bit latch. Thus, during the entire interrupt handling process, if peripheral A triggers an interrupt, it will be missed due the CPU clearing the interrupt bit latch.

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 (erasers 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 will all be 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.

By default, when `ESP_INTR_FLAG_SHARED` flag is specified, the interrupt allocator will allocate only priority level 1 interrupts. Use `ESP_INTR_FLAG_SHARED | ESP_INTR_FLAG_LOWMED` to also allow allocating shared interrupts at priority levels 2 and 3.
Though the framework supports this feature, you have to use it very carefully. There usually exist two ways to stop an interrupt from being triggered: disable the source or mask peripheral interrupt status. ESP-IDF only handles enabling and disabling of the source itself, leaving status and mask bits to be handled by users.

Status bits shall either be masked before the handler responsible for it is disabled, or be masked and then properly handled in another enabled interrupt.

Note: Leaving some status bits unhandled without masking them, while disabling the handlers for them, will cause the interrupt(s) to be triggered indefinitely, resulting therefore in a system crash.

Troubleshooting Interrupt Allocation

On most Espressif SoCs, CPU interrupts are a limited resource. Therefore it is possible for a program to run out of CPU interrupts, for example by initializing several peripheral drivers. Typically, this will result in the driver initialization function returning ESP_ERR_NOT_FOUND error code.

If this happens, you can use esp_intr_dump() function to print the list of interrupts along with their status. The output of this function typically looks like this:

```
CPU 0 interrupt status:
Int  Level  Type  Status
 0    1   Level  Reserved
 1    1   Level  Reserved
 2    1   Level  Used: RTC_CORE
 3    1   Level  Used: TG0_LACT_LEVEL
...
```

The columns of the output have the following meaning:

- **Int:** CPU interrupt input number. This is typically not used in software directly, and is provided for reference only.
- **Level:** Interrupt priority (1-7) of the CPU interrupt. This priority is fixed in hardware, and cannot be changed.
- **Type:** Interrupt type (Level or Edge) of the CPU interrupt. This type is fixed in hardware, and cannot be changed.
- **Status:** One of the possible statuses of the interrupt:
  - **Reserved:** The interrupt is reserved either at hardware level, or by one of the parts of ESP-IDF. It can not be allocated using esp_intr_alloc().
  - **Used:** <source>. The interrupt is allocated and connected to a single peripheral.
  - **Shared:** <source1> <source2> ...: The interrupt is allocated and connected to multiple peripherals. See Multiple Handlers Sharing A Source above.
  - **Free:** The interrupt is not allocated and can be used by esp_intr_alloc().
- **Free (not general-use):** The interrupt is not allocated, but is either a high-priority interrupt (priority 4-7) or an edge-triggered interrupt. High-priority interrupts can be allocated using esp_intr_alloc() but requires the handlers to be written in Assembly, see High Priority Interrupts. Edge-triggered low- and medium-priority interrupts can also be allocated using esp_intr_alloc(), but are not used often since most peripheral interrupts are level-triggered.

If you have confirmed that the application is indeed running out of interrupts, a combination of the following suggestions can help resolve the issue:

- Determine the interrupts which can tolerate higher latency, and allocate them using ESP_INTR_FLAG_SHARED flag (optionally ORed with ESP_INTR_FLAG_LOWMED). Using this flag for two or more peripherals will let them use a single interrupt input, and therefore save interrupt inputs for other peripherals. See Multiple Handlers Sharing A Source above.
- Some peripheral driver may default to allocating interrupts with ESP_INTR_FLAG_LEVEL1 flag, so priority 2 and 3 interrupts do not get used by default. If esp_intr_dump() shows that some priority
2 or 3 interrupts are available, try changing the interrupt allocation flags when initializing the driver to `ESP_INTR_FLAG_LEVEL2` or `ESP_INTR_FLAG_LEVEL3`.

- Check if some of the peripheral drivers do not need to be used all the time, and initialize or deinitialize them on demand. This can reduce the number of simultaneously allocated interrupts.

### API Reference

#### Header File

- `components/esp_hw_support/include/esp_intr_types.h`
- This header file can be included with:

```c
#include "esp_intr_types.h"
```

#### Macros

**ESP_INTR_CPU_AFFINITY_TO_CORE_ID** (cpu_affinity)

Convert esp_intr_cpu_affinity_t to CPU core ID.

#### Type Definitions

```c
typedef void (*intr_handler_t)(void *arg)
```

Function prototype for interrupt handler function

```c
typedef struct intr_handle_data_t *intr_handle_t
```

Handle to an interrupt handler

#### Enumerations

```c
enum esp_intr_cpu_affinity_t
```

Interrupt CPU core affinity.

This type specify the CPU core that the peripheral interrupt is connected to.

**Values:**

- **ESP_INTR_CPU_AFFINITY_AUTO**
  
  Install the peripheral interrupt to ANY CPU core, decided by on which CPU the interrupt allocator is running.

- **ESP_INTR_CPU_AFFINITY_0**
  
  Install the peripheral interrupt to CPU core 0.

- **ESP_INTR_CPU_AFFINITY_1**
  
  Install the peripheral interrupt to CPU core 1.

#### Header File

- `components/esp_hw_support/include/esp_intr_alloc.h`
- This header file can be included with:

```c
#include "esp_intr_alloc.h"
```
Chapter 2. API Reference

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.

**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.

**Returns**

ESP_ERR_INVALID_ARG if cpu or intno is invalid ESP_OK otherwise

**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.

**Parameters**

- **intno** – The number of the interrupt (0-31)
- **cpu** – CPU on which the interrupt should be marked as shared (0 or 1)

**Returns**

ESP_ERR_INVALID_ARG if cpu or intno is invalid ESP_OK otherwise

**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.

**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.

**Returns**

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

**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.
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 ORed 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.

Returns 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

```c
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.

```c
int esp_intr_get_cpu (intr_handle_t handle)
```

Get CPU number an interrupt is tied to.

```c
int esp_intr_get_intno (intr_handle_t handle)
```

Get the allocated interrupt for a certain handle.

```c
esp_err_t esp_intr_disable (intr_handle_t handle)
```

Disable the interrupt associated with the handle.

Note:

a. 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.

b. 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.
Chapter 2. API Reference

**Parameters**
- handle - The handle, as obtained by esp_intr_alloc or esp_intr_alloc_intrstatus

**Returns**
- ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

```c
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.

**Parameters**
- handle - The handle, as obtained by esp_intr_alloc or esp_intr_alloc Intrstatus

**Returns**
- ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

```c
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.

**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.

**Returns**
- ESP_ERR_INVALID_ARG if the combination of arguments is invalid. ESP_OK otherwise

```c
void esp_intr_noniram_disable(void)
```
Disable interrupts that aren’t specifically marked as running from IRAM.

```c
void esp_intr_noniram_enable(void)
```
Re-enable interrupts disabled by esp_intr_noniram_disable.

```c
void esp_intr_enable_source(int inum)
```
enable the interrupt source based on its number

**Parameters**
- inum - interrupt number from 0 to 31

```c
void esp_intr_disable_source(int inum)
```
disable the interrupt source based on its number

**Parameters**
- inum - interrupt number from 0 to 31

```c
static inline int esp_intr_flags_to_level(int flags)
```
Get the lowest interrupt level from the flags.

**Parameters**
- flags - The same flags that pass to esp_intr_alloc_intrstatus API

```c
static inline int esp_intr_level_to_flags(int level)
```
Get the interrupt flags from the supplied level (priority)

**Parameters**
- level - The interrupt priority level

```c
esp_err_t esp_intr_dump(FILE *stream)
```
Dump the status of allocated interrupts.

**Parameters**
- stream - The stream to dump to, if NULL then stdout is used

**Returns**
- ESP_OK on success
**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 pri 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**
- Platform 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
Platform timer 1 interrupt source.

ETS_INTERNAL_TIMER2_INTR_SOURCE
Platform 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_UNUSED_INTR_SOURCE
Interrupt is not assigned to any source.

ETS_INTERNAL_INTR_SOURCE_OFF
Provides SystemView with positive IRQ IDs, otherwise scheduler events are not shown properly

ESP_INTR_ENABLE (inum)
Enable interrupt by interrupt number

ESP_INTR_DISABLE (inum)
Disable interrupt by interrupt number

2.9.21 Logging library

Overview

The logging library provides three ways for setting log verbosity:

- **At compile time**: in menuconfig, set the verbosity level using the option `CONFIG_LOG_DEFAULT_LEVEL`. Optionally, also in menuconfig, set the maximum verbosity level using the option `CONFIG_LOG_MAXIMUM_LEVEL`. By default, this is the same as the default level, but it can be set higher in order to compile more optional logs into the firmware.

- **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.

- **At runtime**: if `CONFIG_LOG_MASTER_LEVEL` is enabled then a Master logging level can be set using `esp_log_set_level_master()`. This option adds an additional logging level check for all compiled logs. Note that this will increase application size. This feature is useful if you want to compile in a lot of logs that are selectable at runtime, but also want to avoid the performance hit from looking up the tags and their log level when you don’t want log output.

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_MAXIMUM_LEVEL`. To increase log level for a specific file above this maximum 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:

```c
static const char* TAG = "MyModule";
```

Then use one of the logging macros to produce output, e.g.:

```c
ESP_LOGW(TAG, "Baud rate error %1.1f%. 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.

There are also `ESP_DRAM_LOGx` versions for each of these macros, e.g. `ESP_DRAM_LOGE`. These versions are used in some places where logging may occur with interrupts disabled or with flash cache inaccessible. Use of this macros should be as sparing as possible, as logging in these types of code should be avoided for performance reasons.

**Note:** Inside critical sections interrupts are disabled so it’s only possible to use `ESP_DRAM_LOGx` (preferred) or `ESP_EARLY_LOGx`. Even though it’s possible to log in these situations, it’s better if your program can be structured not to require it.

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.:

```c
#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include "esp_log.h"
```

At component scope, define it in the component CMakeLists:

```cmake
target_compile_definitions(${COMPONENT_LIB} PUBLIC "-DLOG_LOCAL_LEVEL=ESP_LOG_VERBOSE")
```

To configure logging output per module at runtime, add calls to the function `esp_log_level_set()` as follows:

```c
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
```

**Note:** The “DRAM” and “EARLY” log macro variants documented above do not support per module setting of log verbosity. These macros will always log at the “default” verbosity level, which can only be changed at runtime by calling `esp_log_level("*", level)`.
Even when logs are disabled by using a tag name they will still require a processing time of around 10.9 microseconds per entry.

**Master Logging Level** To enable the Master logging level feature, the `CONFIG_LOG_MASTER_LEVEL` option must be enabled. It adds an additional level check for ESP_LOGx macros before calling `esp_log_write()`. This allows to set a higher `CONFIG_LOG_MAXIMUM_LEVEL`, but not inflict a performance hit during normal operation (only when directed). An application may set the master logging level (`esp_log_set_level_master()`) globally to enforce a maximum log level. ESP_LOGx macros above this level will be skipped immediately, rather than calling `esp_log_write()` and doing a tag lookup. It is recommended to only use this in a top-level application and not in shared components as this would override the global log level for any user using the component. By default, at startup, the Master logging level is `CONFIG_LOG_DEFAULT_LEVEL`.

Note that this feature increases application size because the additional check is added into all ESP_LOGx macros.

The snippet below shows how it works. Setting the Master logging level to ESP_LOG_NONE disables all logging globally. `esp_log_level_set()` does not currently affect logging. But after the Master logging level is released, the logs will be printed as set by `esp_log_level_set()`.

```c
// Master logging level is CONFIG_LOG_DEFAULT_LEVEL at start up and = ESP_LOG_INFO
ESP_LOGI("lib_name", "Message for print"); // prints a INFO message
esp_log_level_set("lib_name", ESP_LOG_WARN); // enables WARN logs from lib_name

esp_log_set_level_master(ESP_LOG_NONE); // disables all logs globally.
// esp_log_level_set has no effect at the moment.

ESP_LOGW("lib_name", "Message for print"); // no print, Master logging level blocks it
esp_log_level_set("lib_name", ESP_LOG_INFO); // enable INFO logs from lib_name

ESP_LOGI("lib_name", "Message for print"); // no print, Master logging level blocks it
esp_log_set_level_master(ESP_LOG_INFO); // enables all INFO logs globally.

ESP_LOGI("lib_name", "Message for print"); // prints a INFO message
```

**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*.

**Thread Safety** The log string is first written into a memory buffer and then sent to the UART for printing. Log calls are thread-safe, i.e., logs of different threads do not conflict with each other.

**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**

Espressif Systems 1638 Release master
Chapter 2. API Reference

- components/log/include/esp_log.h
- This header file can be included with:

```c
#include "esp_log.h"
```

### Functions

**void esp_log_set_level_master (esp_log_level_t level)**

Master log level.

Optional master log level to check against for ESP_LOGx macros before calling esp_log_write. Allows one to set a higher CONFIG_LOG_MAXIMUM_LEVEL but not impose a performance hit during normal operation (only when instructed). An application may set esp_log_set_level_master(level) to globally enforce a maximum log level. ESP_LOGx macros above this level will be skipped immediately, rather than calling esp_log_write and doing a cache hit.

The tradeoff is increased application size.

**Parameters**

- `level` – Master log level

**esp_log_level_t esp_log_get_level_master (void)**

Returns master log level.

**Returns**

Master log level

**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:** Note that this function can not raise log level above the level set using CONFIG_LOG_MAXIMUM_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.

**esp_log_level_t esp_log_level_get (const char *tag)**

Get log level for a given tag, can be used to avoid expensive log statements.

**Parameters**

- `tag` – Tag of the log to query current level. Must be a non-NULL zero terminated string.

**Returns**

The current log level for the given tag

**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.

**Note:** Please note that function callback here must be re-entrant as it can be invoked in parallel from multiple thread context.

**Parameters**

- `func` – new Function used for output. Must have same signature as vprintf.
Returns func old Function used for output.

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.

Returns timestamp, in millisecond

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. Wi-Fi & Bluetooth libraries), these will still
print the RTOS tick time.

Returns 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

Returns 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 also:

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 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 recommended to use terminals with over 102 text width.

See also:

```c
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_HEX** (tag, buffer, buff_len)

Log a buffer of hex bytes at Info level.

See also:

```c
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 also:

```c
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 also:

```c
printf,ESP_LOGE,ESP_DRAM_LOGE
```

In the future, we want to become compatible with clang. Hence, we provide two versions of the following macros which are using variadic arguments. The first one is using the GNU extension ##__VA_ARGS__. The second one is using the C++20 feature VA_OPT(.). This allows
users to compile their code with standard C++20 enabled instead of the GNU extension. Below C++20, we haven’t found any good alternative to using `##VA_ARGS##`.

**ESP_EARLY_LOGW** (tag, format, ...)

macro to output logs in startup code at ESP_LOG_WARN level.

See also:

ESP_EARLY_LOGE, ESP_LOGE, printf

**ESP_EARLY_LOGI** (tag, format, ...)

macro to output logs in startup code at ESP_LOG_INFO level.

See also:

ESP_EARLY_LOGE, ESP_LOGE, printf

**ESP_EARLY_LOGD** (tag, format, ...)

macro to output logs in startup code at ESP_LOG_DEBUG level.

See also:

ESP_EARLY_LOGE, ESP_LOGE, printf

**ESP_EARLY_LOGV** (tag, format, ...)

macro to output logs in startup code at ESP_LOG_VERBOSE level.

See also:

ESP_EARLY_LOGE, ESP_LOGE, printf

**ESP_LOG_EARLY_ENABLED** (log_level)

**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** (level, tag, format, ...)

runtime macro to output logs at a specified level.

See also:

printf

**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
**ESP_LOG_LEVEL_LOCAL** (level, tag, format, ...)

Runtime macro to output logs at a specified level. Also check the level with **LOG_LOCAL_LEVEL**. If **CONFIG_LOG_MASTER_LEVEL** set, also check first against **esp_log_get_level_master()**.

**See also:**

`printf, ESP_LOG_LEVEL`

**ESP_DRAM_LOGE** (tag, format, ...)

Macro to output logs when the cache is disabled. Log at **ESP_LOG_ERROR** level.

Similar to

Usage: `ESP_DRAM_LOGE(DRAM_STR("my_tag"), "format", or ESP_DRAM_LOGE(TAG, “format” , ...)`, where TAG is a char* that points to a str in the DRAM.

**See also:**

`ESP_EARLY_LOGE`, the log level cannot be changed per-tag, however **esp_log_level_set( “*” , level)** will set the default level which controls these log lines also.

**See also:**

`esp_rom_printf, ESP_LOGE`

**Note:** Unlike normal logging macros, it’s possible to use this macro when interrupts are disabled or inside an ISR.

**Note:** Placing log strings in DRAM reduces available DRAM, so only use when absolutely essential.

**ESP_DRAM_LOGW** (tag, format, ...)

Macro to output logs when the cache is disabled at **ESP_LOG_WARN** level.

**See also:**

`ESP_DRAM_LOGW,ESP_LOGW, esp_rom_printf`

**ESP_DRAM_LOGI** (tag, format, ...)

Macro to output logs when the cache is disabled at **ESP_LOG_INFO** level.

**See also:**

`ESP_DRAM_LOGI,ESP_LOGI, esp_rom_printf`

**ESP_DRAM_LOGD** (tag, format, ...)

Macro to output logs when the cache is disabled at **ESP_LOG_DEBUG** level.

**See also:**

`ESP_DRAM_LOGD,ESP_LOGD, esp_rom_printf`
**ESP_DRAM_LOGV** (tag, format, ...)  
macro to output logs when the cache is disabled at ESP_LOG_VERBOSE level.

See also:  
ESP_DRAM_LOGV, ESP_LOGV, esp_rom_printf

**Type Definitions**

typedef int (*vprintf_like_t)(const char*, va_list)

**Enumerations**

enum esp_log_level_t  
Log level.  
Values:

- enumerator **ESP_LOG_NONE**  
  No log output

- enumerator **ESP_LOG_ERROR**  
  Critical errors, software module can not recover on its own

- enumerator **ESP_LOG_WARN**  
  Error conditions from which recovery measures have been taken

- enumerator **ESP_LOG_INFO**  
  Information messages which describe normal flow of events

- enumerator **ESP_LOG_DEBUG**  
  Extra information which is not necessary for normal use (values, pointers, sizes, etc).

- enumerator **ESP_LOG_VERBOSE**  
  Bigger chunks of debugging information, or frequent messages which can potentially flood the output.

### 2.9.22 Miscellaneous System APIs

**Software Reset**

To perform software reset of the chip, the `esp_restart()` function is provided. When the function is called, execution of the program stops, the CPU is reset, the application is loaded by the bootloader and starts execution again.

Additionally, the `esp_register_shutdown_handler()` function can register a routine that will be automatically called before a restart (that is triggered by `esp_restart()`) occurs. This is similar to the functionality of `atexit` POSIX function.
Reset Reason

ESP-IDF applications 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 has ever been available (i.e., the smallest size of free heap memory in the application’s lifetime).

Note that ESP-IDF supports multiple heaps with different capabilities. The functions mentioned in this section return the size of heap memory that can be allocated using the `malloc` family of functions. For further information about heap memory, see Heap Memory Allocation.

MAC Address

These APIs allow querying and customizing MAC addresses for different supported network interfaces (e.g., Wi-Fi, Bluetooth, Ethernet).

To fetch the MAC address for a specific network interface (e.g., Wi-Fi, Bluetooth, Ethernet), call the function `esp_read_mac()`.

In ESP-IDF, the MAC addresses for the various network interfaces are calculated from a single base MAC address. By default, the Espressif base MAC address is used. This base MAC address is pre-programmed into the ESP32-S2 eFuse in the factory during production.

<table>
<thead>
<tr>
<th>Interface</th>
<th>MAC Address (2 universally administered, default)</th>
<th>MAC Address (1 universally administered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wi-Fi Station</td>
<td>base_mac</td>
<td>base_mac</td>
</tr>
<tr>
<td>Wi-Fi SoftAP</td>
<td>base_mac, +1 to the last octet</td>
<td><code>Local MAC</code> (derived from Wi-Fi Station MAC)</td>
</tr>
<tr>
<td>Ethernet</td>
<td><code>Local MAC</code> (derived from Wi-Fi SoftAP MAC)</td>
<td><code>Local MAC</code> (derived from base_mac with +1 to last octet. Not recommended.)</td>
</tr>
</tbody>
</table>

**Note:** The configuration configures the number of universally administered MAC addresses that are provided by Espressif.

**Note:** Although ESP32-S2 has no integrated Ethernet MAC, it is still possible to calculate an Ethernet MAC address. However, this MAC address can only be used with an external ethernet interface such as an SPI-Ethernet device. See Ethernet.

**Custom Interface MAC** Sometimes you may need to define custom MAC addresses that are not generated from the base MAC address. To set a custom interface MAC address, use the `esp_iface_mac_addr_set()` function. This function allows you to overwrite the MAC addresses of interfaces set (or not yet set) by the base MAC address. Once a MAC address has been set for a particular interface, it will not be affected when the base MAC address is changed.
**Custom Base MAC**  The default base MAC is pre-programmed by Espressif in eFuse BLK1. To set a custom base MAC instead, call the function `esp_iface_mac_addr_set()` with the `ESP_MAC_BASE` argument (or `esp_base_mac_addr_set()`) before initializing any network interfaces or calling the `esp_read_mac()` function. The custom MAC address can be stored in any supported storage device (e.g., flash, NVS).

The custom base MAC addresses should be allocated such that derived MAC addresses will not overlap. Based on the table above, users can configure the option `CONFIG_ESP32S2_UNIVERSAL_MAC_ADDRESSES` to set the number of valid universal MAC addresses that can be derived from the custom base MAC.

**Note:** It is also possible to call the function `esp_netif_set_mac()` to set the specific MAC used by a network interface after network initialization. But it is recommended to use the base MAC approach documented here to avoid the possibility of the original MAC address briefly appearing on the network before being changed.

---

**Custom MAC Address in eFuse**  When reading custom MAC addresses from eFuse, ESP-IDF provides a helper function `esp_efuse_mac_get_custom()`. Users can also use `esp_read_mac()` with the `ESP_MAC_EFUSE_CUSTOM` argument. This loads the MAC address from eFuse BLK3. The `esp_efuse_mac_get_custom()` function assumes that the custom base MAC address is stored in the following format:

<table>
<thead>
<tr>
<th>Field</th>
<th># of bits</th>
<th>Range of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC address</td>
<td>48</td>
<td>200:248</td>
</tr>
</tbody>
</table>

**Note:** The eFuse BLK3 uses RS-coding during burning, which means that all eFuse fields in this block must be burnt at the same time.

Once custom eFuse MAC address has been obtained (using `esp_efuse_mac_get_custom()` or `esp_read_mac()`), you need to set it as the base MAC address. There are two ways to do it:

1. Use an old API: call `esp_base_mac_addr_set()`.
2. Use a new API: call `esp_iface_mac_addr_set()` with the `ESP_MAC_BASE` argument.

**Local Versus Universal MAC Addresses**  ESP32-S2 comes pre-programmed with enough valid Espressif universally administered MAC addresses for all internal interfaces. The table above shows how to calculate and derive the MAC address for a specific interface according to the base MAC address.

When using a custom MAC address scheme, it is possible that not all interfaces can be assigned with a universally administered MAC address. In these cases, a locally administered MAC address is assigned. Note that these addresses are intended for use on a single local network only.

See this article for the definition of locally and universally administered MAC addresses.

Function `esp_derive_local_mac()` is called internally to derive a local MAC address from a universal MAC address. The process is as follows:

1. The U/L bit (bit value 0x2) is set in the first octet of the universal MAC address, creating a local MAC address.
2. If this bit is already set in the supplied universal MAC address (i.e., the supplied “universal” MAC address was in fact already a local MAC address), then the first octet of the local MAC address is XORed with 0x4.

**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 ESP-IDF version which is 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 the ESP-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:

```c
#include "esp_idf_version.h"

#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 0, 0)
    // enable functionality present in ESP-IDF v4.0
#endif
```

App Version

The 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 type of the field version is string and it has a maximum length of 32 chars.

To set the version in your project manually, you need to set the `PROJECT_VER` variable in the CMakeLists.txt of your project. In application CMakeLists.txt, put `set(PROJECT_VER "0.1.0.1")` before including `project.cmake`.

If the `CONFIG_APP_PROJECT_VER_FROM_CONFIG` option is set, the value of `CONFIG_APP_PROJECT_VER` will be used. Otherwise, if the `PROJECT_VER` variable is not set in the project, it will be retrieved either from the `$(PROJECT_PATH)/version.txt` file (if present) or using `git` command `git describe`. If neither is available, `PROJECT_VER` will be set to “1”. Application can make use of this by calling `esp_app_get_description()` or `esp_ota_get_partition_description()` functions.

API Reference

Header File

- `components/esp_system/include/esp_system.h`
- This header file can be included with:

```c
#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.

**Parameters**

- `handle` – function to execute on restart

**Returns**

- `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
**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 Wi-Fi, 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.

**Returns** 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:** Note that the returned value may be larger than the maximum contiguous block which can be allocated.

**Returns** Available heap size, in bytes.

**uint32_t esp_get_free_internal_heap_size (void)**

Get the size of available internal heap.

**Note:** Note that the returned value may be larger than the maximum contiguous block which can be allocated.

**Returns** Available internal heap size, in bytes.

**uint32_t esp_get_minimum_free_heap_size (void)**

Get the minimum heap that has ever been available.

**Returns** Minimum free heap ever available

**void esp_system_abort (const char *details)**

Trigger a software abort.

**Parameters**

- **details** – Details that will be displayed during panic handling.

**Type Definitions**

typedef void (*shutdown_handler_t)(void)

Shutdown handler type

**Enumerations**

enum esp_reset_reason_t

Reset reasons.

**Values:**
enumerator **ESP_RST_UNKNOWN**
   Reset reason can not be determined.

enumerator **ESP_RST_POWERON**
   Reset due to power-on event.

enumerator **ESP_RST_EXT**
   Reset by external pin (not applicable for ESP32)

enumerator **ESP_RST_SW**
   Software reset via esp_restart.

enumerator **ESP_RST_PANIC**
   Software reset due to exception/panic.

enumerator **ESP_RST_INT_WDT**
   Reset (software or hardware) due to interrupt watchdog.

enumerator **ESP_RST_TASK_WDT**
   Reset due to task watchdog.

enumerator **ESP_RST_WDT**
   Reset due to other watchdogs.

enumerator **ESP_RST_DEEPSLEEP**
   Reset after exiting deep sleep mode.

enumerator **ESP_RST_BROWNOUT**
   Brownout reset (software or hardware)

enumerator **ESP_RST_SDIO**
   Reset over SDIO.

enumerator **ESP_RST_USB**
   Reset by USB peripheral.

enumerator **ESP_RST_JTAG**
   Reset by JTAG.

**Header File**

- components/esp_common/include/esp_idf_version.h
- This header file can be included with:

```c
#include "esp_idf_version.h"
```

**Functions**
Chapter 2. API Reference

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.

**Returns** 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)

**Header File**

- components/esp_hw_support/include/esp_mac.h
- This header file can be included with:

```c
#include "esp_mac.h"
```

**Functions**

**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 network interfaces.

If using a custom base MAC address, call this API before initializing any network interfaces. Refer to the ESP-IDF Programming Guide for details about how the Base MAC is used.

**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.

**Parameters**
- `mac` - base MAC address, length: 6 bytes. length: 6 bytes for MAC-48
**Returns** ESP_OK on success ESP_ERR_INVALID_ARG If mac is NULL or is not a unicast MAC

*esp_err_t* `esp_base_mac_addr_get` (uint8_t* mac)

Return base MAC address which is set using `esp_base_mac_addr_set`.

**Note:** If no custom Base MAC has been set, this returns the pre-programmed Espressif base MAC address.

**Parameters**

- `mac` - base MAC address, length: 6 bytes. length: 6 bytes for MAC-48

**Returns** ESP_OK on success ESP_ERR_INVALID_ARG mac is NULL ESP_ERR_INVALID_MAC base MAC address has not been set

*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 EFUSE BLK3 in a specified format.

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.

**Note:** This function is currently only supported on ESP32.

**Parameters**

- `mac` - base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64 (used for IEEE 802.15.4, if CONFIG_SOC_IEEE802154_SUPPORTED=y)

**Returns** ESP_OK on success ESP_ERR_INVALID_ARG mac is NULL ESP_ERR_INVALID_MAC CUSTOM_MAC address has not been set ESP_ERR_INVALID_VERSION An invalid MAC version field was read from BLK3 of EFUSE (for esp32) ESP_ERR_INVALID_CRC An invalid MAC CRC was read from BLK3 of EFUSE (for esp32)

*esp_err_t* `esp_efuse_mac_get_default` (uint8_t* mac)

Return base MAC address which is factory-programmed by Espressif in EFUSE.

**Parameters**

- `mac` - base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64 (used for IEEE 802.15.4, if CONFIG_SOC_IEEE802154_SUPPORTED=y)

**Returns** ESP_OK on success ESP_ERR_INVALID_ARG mac is NULL

*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()`. Then calculates the MAC address of the specific interface requested, refer to ESP-IDF Programming Guide for the algorithm.

The MAC address set by the `esp iface_mac_addr_set()` function will not depend on the base MAC address.

**Parameters**

- `mac` - base MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64 (used for IEEE 802.15.4, if CONFIG_SOC_IEEE802154_SUPPORTED=y)
- `type` - Type of MAC address to return

**Returns** ESP_OK on success

*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 copies a universal MAC address and then sets the “locally administered” bit (bit 0x2) in the first octet, creating a locally administered MAC address.
If the universal MAC address argument is already a locally administered MAC address, then the first octet is XORed with 0x4 in order to create a different locally administered MAC address.

**Parameters**
- `local_mac` – base MAC address, length: 6 bytes. length: 6 bytes for MAC-48
- `universal_mac` – Source universal MAC address, length: 6 bytes.

**Returns** ESP_OK on success

`esp_err_t esp_iface_mac_addr_set(const uint8_t*mac, esp_mac_type_t type)`
Set custom MAC address of the interface. This function allows you to overwrite the MAC addresses of the interfaces set by the base MAC address.

**Parameters**
- `mac` – MAC address, length: 6 bytes/8 bytes. length: 6 bytes for MAC-48 8 bytes for EUI-64 (used for ESP_MAC_IEEE802154 type, if CONFIG_SOC_IEEE802154_SUPPORTED=y)
- `type` – Type of MAC address

**Returns** ESP_OK on success

`size_t esp_mac_addr_len_get(esp_mac_type_t type)`
Return the size of the MAC type in bytes.

If CONFIG_SOC_IEEE802154_SUPPORTED is set then for these types:
- ESP_MAC_IEEE802154 is 8 bytes.
- ESP_MAC_BASE, ESP_MAC_EFUSE_FACTORY and ESP_MAC_EFUSE_CUSTOM the MAC size is 6 bytes.
- ESP_MAC_EFUSE_EXT is 2 bytes. If CONFIG_SOC_IEEE802154_SUPPORTED is not set then for all types it returns 6 bytes.

**Parameters** `type` – Type of MAC address

**Returns** 0 MAC type not found (not supported) 6 bytes for MAC-48, 8 bytes for EUI-64.

**Macros**

`MAC2STR(a)`

`MACSTR`

**Enumerations**

`enum esp_mac_type_t`

**Values:**
- `ESP_MAC_WIFI_STA` – MAC for WiFi Station (6 bytes)
- `ESP_MAC_WIFI_SOFTAP` – MAC for WiFi Soft-AP (6 bytes)
- `ESP_MAC_BT` – MAC for Bluetooth (6 bytes)
- `ESP_MAC_ETH` – MAC for Ethernet (6 bytes)
enumerator ESP_MAC_IEEE802154
    if CONFIG_SOC_IEEE802154_SUPPORTED=y, MAC for IEEE802154 (8 bytes)

enumerator ESP_MAC_BASE
    Base MAC for that used for other MAC types (6 bytes)

enumerator ESP_MAC_EFUSE_FACTORY
    MAC_FACTORY eFuse which was burned by Espressif in production (6 bytes)

enumerator ESP_MAC_EFUSE_CUSTOM
    MAC_CUSTOM eFuse which was can be burned by customer (6 bytes)

enumerator ESP_MAC_EFUSE_EXT
    if CONFIG_SOC_IEEE802154_SUPPORTED=y, MAC_EXT eFuse which is used as an extender for IEEE802154 MAC (2 bytes)

Header File

- components/esp_hw_support/include/esp_chip_info.h
- This header file can be included with:

```c
#include "esp_chip_info.h"
```

Functions

void esp_chip_info (esp_chip_info_t *out_info)
    Fill an esp_chip_info_t structure with information about the chip.

    Parameters out_info -[out] 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

uint16_t revision
    chip revision number (in format MXX; where M - wafer major version, XX - wafer minor version)

uint8_t cores
    number of CPU cores
```
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.

CHIP_FEATURE_IEEE802154
    Chip has IEEE 802.15.4.

CHIP_FEATURE_EMB_PSRAM
    Chip has embedded psram.

Enumerations

enum esp_chip_model_t
    Chip models.
    Values:

        enumerator CHIP_ESP32
            ESP32.
        enumerator CHIP_ESP32S2
            ESP32-S2.
        enumerator CHIP_ESP32S3
            ESP32-S3.
        enumerator CHIP_ESP32C3
            ESP32-C3.
        enumerator CHIP_ESP32C2
            ESP32-C2.
        enumerator CHIP_ESP32C6
            ESP32-C6.
        enumerator CHIP_ESP32H2
            ESP32-H2.
        enumerator CHIP_ESP32P4
            ESP32-P4.
Chapter 2. API Reference

enumerator CHIP_POSIX_LINUX
The code is running on POSIX/Linux simulator.

Header File

- components/esp_hw_support/include/esp_cpu.h
- This header file can be included with:

```
#include "esp_cpu.h"
```

Functions

void esp_cpu_stall (int core_id)
Stall a CPU core.

Parameters
- core_id - The core’s ID

void esp_cpu_unstall (int core_id)
Resume a previously stalled CPU core.

Parameters
- core_id - The core’s ID

void esp_cpu_reset (int core_id)
Reset a CPU core.

Parameters
- core_id - The core’s ID

void esp_cpu_wait_for_intr (void)
Wait for Interrupt.

This function causes the current CPU core to execute its Wait For Interrupt (WFI or equivalent) instruction. After executing this function, the CPU core will stop execution until an interrupt occurs.

int esp_cpu_get_core_id (void)
Get the current core’s ID.

This function will return the ID of the current CPU (i.e., the CPU that calls this function).

Returns
- The current core’s ID [0..SOC_CPU_CORES_NUM - 1]

void *esp_cpu_get_sp (void)
Read the current stack pointer address.

Returns
- Stack pointer address

esp_cpu_cycle_count_t esp_cpu_get_cycle_count (void)
Get the current CPU core’s cycle count.

Each CPU core maintains an internal counter (i.e., cycle count) that increments every CPU clock cycle.

Returns
- Current CPU’s cycle count, 0 if not supported.

void esp_cpu_set_cycle_count (esp_cpu_cycle_count_t cycle_count)
Set the current CPU core’s cycle count.

Set the given value into the internal counter that increments every CPU clock cycle.

Parameters
- cycle_count - CPU cycle count

void *esp_cpu_pc_to_addr (uint32_t pc)
Convert a program counter (PC) value to address.

If the architecture does not store the true virtual address in the CPU’s PC or return addresses, this function will convert the PC value to a virtual address. Otherwise, the PC is just returned.

Parameters
- pc - PC value
Returns Virtual address

```c
void esp_cpu_intr_get_desc (int core_id, int intr_num, esp_cpu_intr_desc_t *intr_desc_ret)
```

Get a CPU interrupt’s descriptor.

Each CPU interrupt has a descriptor describing the interrupt’s capabilities and restrictions. This function gets the descriptor of a particular interrupt on a particular CPU.

**Parameters**
- `core_id` - [in] The core’s ID
- `intr_num` - [in] Interrupt number
- `intr_desc_ret` - [out] The interrupt’s descriptor

```c
void esp_cpu_intr_set_ivt_addr (const void *ivt_addr)
```

Set the base address of the current CPU’s Interrupt Vector Table (IVT)

**Parameters**
- `ivt_addr` - Interrupt Vector Table’s base address

```c
bool esp_cpu_intr_has_handler (int intr_num)
```

Check if a particular interrupt already has a handler function.

Check if a particular interrupt on the current CPU already has a handler function assigned.

**Note:** This function simply checks if the IVT of the current CPU already has a handler assigned.

**Parameters**
- `intr_num` - Interrupt number (from 0 to 31)

**Returns** True if the interrupt has a handler function, false otherwise.

```c
void esp_cpu_intr_set_handler (int intr_num, esp_cpu_intr_handler_t handler, void *handler_arg)
```

Set the handler function of a particular interrupt.

Assign a handler function (i.e., ISR) to a particular interrupt on the current CPU.

**Note:** This function simply sets the handler function (in the IVT) and does not actually enable the interrupt.

**Parameters**
- `intr_num` - Interrupt number (from 0 to 31)
- `handler` - Handler function
- `handler_arg` - Argument passed to the handler function

```c
void *esp_cpu_intr_get_handler_arg (int intr_num)
```

Get a handler function’s argument of.

Get the argument of a previously assigned handler function on the current CPU.

**Parameters**
- `intr_num` - Interrupt number (from 0 to 31)

**Returns** The the argument passed to the handler function

```c
void esp_cpu_intr_enable (uint32_t intr_mask)
```

Enable particular interrupts on the current CPU.

**Parameters**
- `intr_mask` - Bit mask of the interrupts to enable

```c
void esp_cpu_intr_disable (uint32_t intr_mask)
```

Disable particular interrupts on the current CPU.

**Parameters**
- `intr_mask` - Bit mask of the interrupts to disable
Chapter 2. API Reference

```c
uint32_t esp_cpu_intr_get_enabled_mask (void)

Get the enabled interrupts on the current CPU.

  Returns  Bit mask of the enabled interrupts
```

```c
void esp_cpu_intr_edge_ack (int intr_num)

Acknowledge an edge interrupt.

  Parameters  intr_num – Interrupt number (from 0 to 31)
```

```c
void esp_cpu_configure_region_protection (void)

Configure the CPU to disable access to invalid memory regions.
```

```c
esp_err_t esp_cpu_set_breakpoint (int bp_num, const void *bp_addr)

Set and enable a hardware breakpoint on the current CPU.

  Note: This function is meant to be called by the panic handler to set a breakpoint for an attached debugger during a panic.

  Parameters  bp_num – Hardware breakpoint number [0..SOC_CPU_BREAKPOINTS_NUM - 1]
  bp_addr – Address to set a breakpoint on

  Returns  ESP_OK if breakpoint is set. Failure otherwise
```

```c
esp_err_t esp_cpu_clear_breakpoint (int bp_num)

Clear a hardware breakpoint on the current CPU.

  Parameters  bp_num – Hardware breakpoint number [0..SOC_CPU_BREAKPOINTS_NUM - 1]

  Returns  ESP_OK if breakpoint is cleared. Failure otherwise
```

```c
esp_err_t esp_cpu_set_watchpoint (int wp_num, const void *wp_addr, size_t size, esp_cpu_watchpoint_trigger_t trigger)

Set and enable a hardware watchpoint on the current CPU.

  Parameters  wp_num – Hardware watchpoint number [0..SOC_CPU_WATCHPOINTS_NUM - 1]
  wp_addr – Watchpoint’s base address, must be naturally aligned to the size of the region
  size – Size of the region to watch. Must be one of 2^n and in the range of [1 .. SOC_CPU_WATCHPOINT_MAX_REGION_SIZE]
```

Note: Overwrites previously set watchpoint with same watchpoint number.

Note: This function is meant to be called by the panic handler to set a breakpoint for an attached debugger during a panic.

Note: Clears a breakpoint regardless of whether it was previously set.

Note: Overwrites previously set watchpoint with same watchpoint number. On RISC-V chips, this API uses method0(Exact matching) and method1(NAPOT matching) according to the riscv-debug-spec-0.13 specification for address matching. If the watch region size is 1 byte, it uses exact matching (method 0). If the watch region size is larger than 1 byte, it uses NAPOT matching (method 1). This mode requires the watching region start address to be aligned to the watching region size.
Chapter 2. API Reference

- **trigger** – Trigger type
  
  *Returns* ESP_ERR_INVALID_ARG on invalid arg, ESP_OK otherwise

```c
esp_err_t esp_cpu_clear_watchpoint (int wp_num)
```

Clear a hardware watchpoint on the current CPU.

**Note:** Clears a watchpoint regardless of whether it was previously set

**Parameters**
- **wp_num** – Hardware watchpoint number [0..SOC_CPU_WATCHPOINTS_NUM - 1]
  
  *Returns* ESP_OK if watchpoint was cleared. Failure otherwise.

```c
bool esp_cpu_dbgr_is_attached (void)
```

Check if the current CPU has a debugger attached.

*Returns* True if debugger is attached, false otherwise

```c
void esp_cpu_dbgr_break (void)
```

Trigger a call to the current CPU’s attached debugger.

```c
intptr_t esp_cpu_get_call_addr (intptr_t return_address)
```

Given the return address, calculate the address of the preceding call instruction. This is typically used to answer the question “where was the function called from?”

**Parameters**
- **return_address** – The value of the return address register. Typically set to the value of __builtin_return_address(0).
  
  *Returns* Address of the call instruction preceding the return address.

```c
bool esp_cpu_compare_and_set (volatile uint32_t *addr, uint32_t compare_value, uint32_t new_value)
```

Atomic compare-and-set operation.

**Parameters**
- **addr** – Address of atomic variable
- **compare_value** – Value to compare the atomic variable to
- **new_value** – New value to set the atomic variable to
  
  *Returns* Whether the atomic variable was set or not

### Structures

```c
struct esp_cpu_intr_desc_t
```

CPU interrupt descriptor.

Each particular CPU interrupt has an associated descriptor describing that particular interrupt’s characteristics. Call esp_cpu_intr_get_desc() to get the descriptors of a particular interrupt.

### Public Members

```c
int priority
```

Priority of the interrupt if it has a fixed priority, (-1) if the priority is configurable.

```c
esp_cpu_intr_type_t type
```

Whether the interrupt is an edge or level type interrupt, ESP_CPU_INTR_TYPE_NA if the type is configurable.

```c
uint32_t flags
```

Flags indicating extra details.
Chapter 2. API Reference

Macros

**ESP_CPU_INTR_DESC_FLAG_SPECIAL**

Interrupt descriptor flags of `esp_cpu_intr_desc_t`.

The interrupt is a special interrupt (e.g., a CPU timer interrupt)

**ESP_CPU_INTR_DESC_FLAG_RESVD**

The interrupt is reserved for internal use

Type Definitions

typedef uint32_t **esp_cpu_cycle_count_t**

CPU cycle count type.

This data type represents the CPU’s clock cycle count

typedef void (*`esp_cpu_intr_handler_t`)(void *arg)

CPU interrupt handler type.

Enumerations

enum **esp_cpu_intr_type_t**

CPU interrupt type.

*Values:*

enumerator **ESP_CPU_INTR_TYPE_LEVEL**

enumerator **ESP_CPU_INTR_TYPE_EDGE**

enumerator **ESP_CPU_INTR_TYPE_NA**

enum **esp_cpu_watchpoint_trigger_t**

CPU watchpoint trigger type.

*Values:*

enumerator **ESP_CPU_WATCHPOINT_LOAD**

enumerator **ESP_CPU_WATCHPOINT_STORE**

enumerator **ESP_CPU_WATCHPOINT_ACCESS**

Header File

- components/esp_app_format/include/esp_app_desc.h
- This header file can be included with:

```c
#include "esp_app_desc.h"
```

- This header file is a part of the API provided by the esp_app_format component. To declare that your component depends on esp_app_format, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_app_format
```

or
Functions

const esp_app_desc_t *esp_app_get_description (void)

Return esp_app_desc structure. This structure includes app version.

Returns Pointer to esp_app_desc structure.

int esp_app_get_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.

Parameters
- dst - Destination buffer
- size - Size of the buffer

Returns Number of bytes written to dst (including null terminator)

char *esp_app_get_elf_sha256_str (void)

Return SHA256 of the ELF file which is already formatted as hexadecimal, null-terminated included. Can be used in panic handler or core dump during when cache is disabled. The length is defined by CONFIG_APP_RETRIEVE_LEN_ELF_SHA option.

Returns Hexadecimal SHA256 string

Structures

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
Chapter 2. API Reference

char date[16]
Compile date

tchar idf_ver[32]
Version IDF

uint8_t app_elf_sha256[32]

sha256 of elf file

uint32_t reserv2[20]

reserv2

Macros

ESP_APP_DESC_MAGIC_WORD
The magic word for the esp_app_desc structure that is in DROM.

2.9.23 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 Wi-Fi or Bluetooth.)

OTA requires configuring the Partition Tables of the device with at least two OTA app slot partitions (i.e., 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 that is currently not selected 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 Tables 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 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 but rather 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:

<table>
<thead>
<tr>
<th>States</th>
<th>Restriction of selecting a boot app in bootloader</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP_OTA_IMG_Valid</td>
<td>Non-restriction. Will be selected.</td>
</tr>
<tr>
<td>ESP_OTA_IMG_Invalid</td>
<td>Non-restriction. Will be selected.</td>
</tr>
<tr>
<td>ESP_OTA_IMG_ABORTED</td>
<td>Will not be selected.</td>
</tr>
<tr>
<td>ESP_OTA_IMG_NEW</td>
<td><code>CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE</code> option is set it will be selected only once. In bootloader the state immediately changes to ESP_OTA_IMG_PENDING_VERIFY.</td>
</tr>
<tr>
<td>ESP_OTA_IMG_PENDING_VERIFY</td>
<td><code>CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE</code> option is set it will not be selected, and the state will change to ESP_OTA_IMG_ABORTED.</td>
</tr>
</tbody>
</table>

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 is 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 working 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, which prevents 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 odata.
• 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 version will be updated. 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 as the device may not have any bootable apps. It will then 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 the 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.

```c
bool image_header_was_checked = false;
while (1) {
    int data_read = esp_http_client_read(client, ota_write_data, BUFFSIZE);
    ...
    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 16 bits. This means that only 16 times you can do an anti-rollback. You can reduce the length of this efuse field using ` CON-`
FIG_BOOTLOADER_APP_SEC_VER_SIZE_EFUSE_FIELD option.

- Factory and Test partitions are not supported in anti-rollback scheme and hence partition table should not have partition with SubType set to factory or test.

security_version:

- In application image it is stored in esp_app_desc structure. The number is set CONFIG_BOOTLOADER_APP_SECURE_VERSION.

Secure OTA Updates Without Secure Boot

The verification of signed OTA updates can be performed even without enabling hardware secure boot. This can be achieved by setting CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT and CONFIG_SECURE_SIGNED_ON_UPDATE_NO_SECURE_BOOT.

OTA Tool otatool.py

The component app_update provides a tool app_update/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)
```

(continues on next page)
# Read OTA partition 'ota_3' and save contents to a file named 'ota_3.bin'
```python
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 Tables
- Partitions API
- SPI Flash API
- ESP HTTPS OTA

## Application Example

End-to-end example of OTA firmware update workflow: `system/ota`.

## API Reference

### Header File

- `components/app_update/include/esp_ota_ops.h`

This header file can be included with:

```c
#include "esp_ota_ops.h"
```
• This header file is a part of the API provided by the `app_update` component. To declare that your component depends on `app_update`, add the following to your CMakeLists.txt:

```cmake
REQUIRES app_update
```
or
```
PRIV_REQUIRES app_update
```

## Functions

```c
const esp_app_desc_t *esp_ota_get_app_description (void)
```

Return `esp_app_desc` structure. This structure includes app version.

**Note:** This API is present for backward compatibility reasons. Alternative function with the same functionality is `esp_app_get_description`.

**Returns** Pointer to `esp_app_desc` structure.

```c
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.

**Note:** This API is present for backward compatibility reasons. Alternative function with the same functionality is `esp_app_get_elf_sha256`.

**Parameters**
- `dst` - Destination buffer
- `size` - Size of the buffer

**Returns** Number of bytes written to `dst` (including null terminator)

```c
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).

**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.

**Returns**
• 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_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.

\texttt{esp_err_t esp_ota_write} (\texttt{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.

\textbf{Parameters}

\begin{itemize}
\item \texttt{handle} - Handle obtained from \texttt{esp_ota_begin}
\item \texttt{data} - Data buffer to write
\item \texttt{size} - Size of data buffer in bytes
\end{itemize}

\textbf{Returns}

\begin{itemize}
\item ESP_OK: Data was written to flash successfully.
\item ESP_ERR_INVALID_ARG: handle is invalid.
\item ESP_ERR_OTA_VALIDATE_FAILED: First byte of image contains invalid app image magic byte.
\item ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
\item ESP_ERR_OTA_SELECT_INFO_INVALID: OTA data partition has invalid contents
\end{itemize}

\texttt{esp_err_t esp_ota_write_with_offset} (\texttt{esp_ota_handle_t handle, const void *data, size_t size, uint32_t offset})

Write OTA update data to partition at an offset.

This function can write data in non-contiguous manner. If flash encryption is enabled, data should be 16 bytes aligned.

\textbf{Note:} While performing OTA, if the packets arrive out of order, \texttt{esp_ota_write_with_offset()} can be used to write data in non-contiguous manner. Use of \texttt{esp_ota_write_with_offset()} in combination with \texttt{esp_ota_write()} is not recommended.

\textbf{Parameters}

\begin{itemize}
\item \texttt{handle} - Handle obtained from \texttt{esp_ota_begin}
\item \texttt{data} - Data buffer to write
\item \texttt{size} - Size of data buffer in bytes
\item \texttt{offset} - Offset in flash partition
\end{itemize}

\textbf{Returns}

\begin{itemize}
\item ESP_OK: Data was written to flash successfully.
\item ESP_ERR_INVALID_ARG: handle is invalid.
\item ESP_ERR_OTA_VALIDATE_FAILED: First byte of image contains invalid app image magic byte.
\item ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
\item ESP_ERR_OTA_SELECT_INFO_INVALID: OTA data partition has invalid contents
\end{itemize}
Chapter 2. API Reference

`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).

**Parameters**
- handle – Handle obtained from `esp_ota_begin()`.

**Returns**
- 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.

`esp_err_t esp_ota_abort(esp_ota_handle_t handle)`
Abort OTA update, free the handle and memory associated with it.

**Parameters**
- handle – obtained from `esp_ota_begin()`.

**Returns**
- ESP_OK: Handle and its associated memory is freed successfully.
- ESP_ERR_NOT_FOUND: OTA handle was not found.

`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.

**Parameters**
- partition – Pointer to info for partition containing app image to boot.

**Returns**
- 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.

`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.
Returns  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.

```c
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.

Returns  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.

```c
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.

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.

Returns  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.

```c
esp_err_t esp_ota_get_partition_description (const esp_partition_t *partition, esp_app_desc_t *app_desc)
Return esp_app_desc structure for app partition. This structure includes app version.
```

Returns a description for the requested app partition.

Parameters  
- `partition` – [in] Pointer to app partition. (only app partition)
- `app_desc` – [out] Structure of info about app.

Returns  
- 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.

```c
esp_err_t esp_ota_get_bootloader_description (const esp_partition_t *bootloader_partition, esp_bootloader_desc_t *desc)
```

Returns the description structure of the bootloader.

Parameters  
- `bootloader_partition` – [in] Pointer to bootloader partition. If NULL, then the current bootloader is used (the default location).
- `desc` – [out] Structure of info about bootloader.

Returns  
- ESP_OK Successful.
- ESP_ERR_NOT_FOUND Description structure is not found in the bootloader image. Magic byte is incorrect.
- ESP_ERR_INVALID_ARG Arguments is NULL.
• ESP_ERR_INVALID_SIZE Read would go out of bounds of the partition.
• or one of error codes from lower-level flash driver.

`uint8_t esp_ota_get_app_partition_count (void)`
Returns number of ota partitions provided in partition table.

Returns
• Number of OTA partitions

`esp_err_t esp_ota_mark_app_valid_cancel_rollback (void)`
This function is called to indicate that the running app is working well.

Returns
• 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.

Returns
• 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).

Returns
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.

Parameters
• partition –[in] Pointer to partition.
• ota_state –[out] state of partition (if this partition has a record in otadata).

Returns
• 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.

`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.

Returns
• 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 >= secure_version of efuse (if anti-rollback is enabled).

Returns
• True: Returns true if the slots have at least one app (except the running app).
• False: The rollback is not possible.
esp_err_t esp_ota_revoke_secure_boot_public_key (esp_ota_secure_boot_public_key_index_t index)

Revoke the signature digest denoted by the given index. This should be called in the application only after the rollback logic otherwise the device may end up in unrecoverable state.

Relevant for Secure boot v2 on ESP32-S2, ESP32-S3, ESP32-C3, ESP32-C6, ESP32-H2 where up to 3 key digests can be stored (Key #N-1, Key #N, Key #N+1). When a key used to sign an app is invalidated, an OTA update is to be sent with an app signed with at least one of the other two keys which has not been revoked already. After successfully booting the OTA app should call this function to revoke Key #N-1.

Parameters

index -- The index of the signature block to be revoked

Returns

• ESP_OK: If revocation is successful.
• ESP_ERR_INVALID_ARG: If the index of the public key to be revoked is incorrect.
• ESP_FAIL: If secure boot v2 has not been enabled.

Macros

OTA_SIZE_UNKNOWN

Used for esp_ota_begin() if new image size is unknown

OTA_WITH_SEQUENTIAL_WRITES

Used for esp_ota_begin() if new image size is unknown and erase can be done in incremental manner (assuming write operation is in continuous sequence)

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().
Enumerations

```c
enum esp_ota_secure_boot_public_key_index_t
{
    SecureBootV2publickeyindexes.
    Values:

    enumerator SECURE_BOOT_PUBLIC_KEY_INDEX_0
        Points to the 0th index of the Secure Boot v2 public key

    enumerator SECURE_BOOT_PUBLIC_KEY_INDEX_1
        Points to the 1st index of the Secure Boot v2 public key

    enumerator SECURE_BOOT_PUBLIC_KEY_INDEX_2
        Points to the 2nd index of the Secure Boot v2 public key
```

Debugging OTA Failure

![Fig. 50: How to Debug When OTA Fails (click to enlarge)](image)

2.9.24 Performance Monitor
The Performance Monitor component provides APIs to use ESP32-S2 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**

- `components/perfmon/include/xtensa_perfmon_access.h`
  
  This header file can be included with:

  ```
  #include "xtensa_perfmon_access.h"
  ```

- This header file is a part of the API provided by the `perfmon` component. To declare that your component depends on `perfmon`, add the following to your `CMakeLists.txt`:

  ```
  REQUIRES perfmon
  ```

  or

  ```
  PRIV_REQUIRES perfmon
  ```

**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

  **Parameters**

  - `id` - [in] performance counter number
  - `select` - [in] select value from PMCTRLx register
  - `mask` - [in] mask value from PMCTRLx register
  - `kernelcnt` - [in] kernelcnt value from PMCTRLx register
  - `tracelevel` - [in] tracelevel value from PMCTRLx register

  **Returns**

  - ESP_OK on success
  - ESP_ERR_INVALID_ARG if one of the arguments is not correct

`esp_err_t xtensa_perfmon_reset` (int id)

- Reset PM counter.

  Reset PM counter. Writes 0 to the PMx register.

  **Parameters** `id` - [in] performance counter number

  **Returns**

  - ESP_OK on success
  - ESP_ERR_INVALID_ARG if id out of range
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.
    Parameters id – [in] performance counter number
    Returns
    • Performance counter value

esp_err_t *xtensa_perfmon_overflow (int id)
    Read PM overflow state.
    Read overflow value of defined PM counter.
    Parameters id – [in] performance counter number
    Returns
    • ESP_OK if there is no overflow (overflow = 0)
    • ESP_FAIL if overflow occure (overflow = 1)

void *xtensa_perfmon_dump (void)
    Dump PM values.
    Dump all PM register to the console.

Header File
• components/perfmon/include/xtensa_perfmon_apis.h
• This header file can be included with:

```
#include "xtensa_perfmon_apis.h"
```

• This header file is a part of the API provided by the perfmon component. To declare that your component depends on perfmon, add the following to your CMakeLists.txt:

```
REQUIRES perfmon
```

or

```
PRIV_REQUIRES perfmon
```

Functions

esp_err_t *xtensa_perfmon_exec (const xtensa_perfmon_config_t *config)
    ExecutePM.
    Execute performance counter for dedicated function with defined parameters
    Parameters config – [in] pointer to the configuration structure
    Returns
    • ESP_OK if no errors
    • ESP_ERR_INVALID_ARG if one of the required parameters not defined
    • ESP_FAIL - counter overflow
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 call-
back_params is set to NULL, will print to stdout

Parameters
• params [in] used parameters passed from configuration (callback_params). This pa-
   rameter expected as FILE* hanle, where data will be stored. If this parameter NULL,
   then data will be stored to the stdout.
• select [in] select value for current counter
• mask [in] mask value for current counter
• value [in] 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
Chapter 2. API Reference

Type Definitions

typedef struct xtensa_perfmon_config xtensa_perfmon_config_t

Performance monitor configuration structure.

Structure to configure performance counter to measure dedicated function

2.9.25 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 has 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_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 is usually set to the default CPU frequency.
• 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_ESP_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-S2 supports three types of locks described in the table below.

<table>
<thead>
<tr>
<th>Lock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP_PM_CPU_FREQ_MAX</td>
<td>Requests CPU frequency to be at the maximum value set with esp_pm_configure(). For ESP32-S2, this value can be set to 80 MHz, 160 MHz, or 240 MHz.</td>
</tr>
<tr>
<td>ESP_PM_APB_FREQ_MAX</td>
<td>Requests the APB frequency to be at the maximum supported value. For ESP32-S2, this is 80 MHz.</td>
</tr>
<tr>
<td>ESP_PM_NO_LIGHT_SLEEP</td>
<td>Disables automatic switching to Light-sleep.</td>
</tr>
</tbody>
</table>

ESP32-S2 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_ESP_DEFAULT_CPU_FREQ_MHZ.
### 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 operation of some peripherals, while other peripherals may have issues. For example, Timer Group peripheral timers keeps counting, however, the speed at which they count changes proportionally to the APB frequency.

Peripheral clock sources such as REF_TICK, XTAL, RC_FAST (i.e., RTC_8M), their frequencies will not be influenced by APB frequency. And therefore, to ensure the peripheral behaves consistently during DFS, it is recommended to select one of these clocks as the peripheral clock source. For more specific guidelines, please refer to the “Power Management” section of each peripheral’s “API Reference > Peripherals API” page.

Currently, the following peripheral drivers are aware of DFS and use the `ESP_PM_APB_FREQ_MAX` lock for the duration of the transaction:

<table>
<thead>
<tr>
<th>Max CPU Frequency Set</th>
<th>Lock Acquisition</th>
<th>CPU and APB Frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td><code>ESP_PM_CPU_FREQ_MAX</code> acquired</td>
<td>CPU: 240 MHz&lt;br&gt;APB: 80 MHz</td>
</tr>
<tr>
<td></td>
<td><code>ESP_PM_APB_FREQ_MAX</code> acquired, <code>ESP_PM_CPU_FREQ_MAX</code> not acquired</td>
<td>CPU: 80 MHz&lt;br&gt;APB: 80 MHz</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>Min values for both frequencies set with <code>esp_pm_configure()</code></td>
</tr>
<tr>
<td>160</td>
<td><code>ESP_PM_CPU_FREQ_MAX</code> acquired</td>
<td>CPU: 160 MHz&lt;br&gt;APB: 80 MHz</td>
</tr>
<tr>
<td></td>
<td><code>ESP_PM_APB_FREQ_MAX</code> acquired, <code>ESP_PM_CPU_FREQ_MAX</code> not acquired</td>
<td>CPU: 80 MHz&lt;br&gt;APB: 80 MHz</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>Min values for both frequencies set with <code>esp_pm_configure()</code></td>
</tr>
<tr>
<td>80</td>
<td>Any of <code>ESP_PM_CPU_FREQ_MAX</code> or <code>ESP_PM_APB_FREQ_MAX</code> acquired</td>
<td>CPU: 80 MHz&lt;br&gt;APB: 80 MHz</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>Min values for both frequencies set with <code>esp_pm_configure()</code></td>
</tr>
</tbody>
</table>

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 is chosen to wake up the chip before the nearest event (task being unblocked, or timer elapses). To skip unnecessary wake-up, you can consider initializing an `esp_timer` with the `skip_unhandled_events` option as `true`. Timers with this flag will not wake up the system and it helps to reduce consumption.
Chapter 2. API Reference

- 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 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()`.
- GPTimer: between calls to `gptimer_enable()` and `gptimer_disable()`.
- 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.
- TWAI: between calls to `twai_driver_install()` and `twai_driver_uninstall()` (only when the clock source is set to TWAI_CLK_SRC_APB).

The following peripheral drivers are not aware of DFS yet. Applications need to acquire/release locks themselves, when necessary:

- PCNT
- Sigma-delta
- The legacy timer group driver

Light-sleep Peripheral Power Down

API Reference

Header File

- components/esp_pm/include/esp_pm.h
- This header file can be included with:

  ```
  #include "esp_pm.h"
  ```
- This header file is a part of the API provided by the esp_pm component. To declare that your component depends on `esp_pm`, add the following to your CMakeLists.txt:

  ```
  REQUIRE esp_pm
  ```
  or
  ```
  PRIV_REQUIRES esp_pm
  ```

Functions

`esp_err_t esp_pm_configure (const void *config)`

Set implementation-specific power management configuration.

- **Parameters** config - pointer to implementation-specific configuration structure (e.g. esp_pm_config_esp32)
- **Returns**
  - 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

`esp_err_t esp_pm_get_configuration (void *config)`

Get implementation-specific power management configuration.

- **Parameters** config - pointer to implementation-specific configuration structure (e.g. esp_pm_config_esp32)
### Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG if the pointer is null

```c
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 a 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.

### 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_handle` - [out] 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.

### Returns

- ESP_OK on success
- 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`

### 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.

#### Parameters

- `handle` - Handle obtained from `esp_pm_lock_create` function

#### Returns

- 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`

### 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 function 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.

#### Parameters

- `handle` - Handle obtained from `esp_pm_lock_create` function

#### Returns

- 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`
**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.

**Parameters**

- **handle** - handle obtained from esp_pm_lock_create function

**Returns**

- 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

**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.

**Parameters**

- **stream** - stream to print information to; use stdout or stderr to print to the console; use fnmopen/open_memstream to print to a string buffer.

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if CONFIG_PM_ENABLE is not enabled in sdkconfig

**Structures**

struct **esp_pm_config_t**

Power management config.

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

**Type Definitions**

typedef **esp_pm_config_t esp_pm_config_esp32_t**

backward compatibility newer chips no longer require this typedef

typedef **esp_pm_config_t esp_pm_config_esp32s2_t**

typedef **esp_pm_config_t esp_pm_config_esp32s3_t**

typedef **esp_pm_config_t esp_pm_config_esp32c3_t**
typedef esp_pm_config_t esp_pm_config_esp32c2_t

typedef esp_pm_config_t esp_pm_config_esp32c6_t

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:

enumerator 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.

enumerator 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.

enumerator ESP_PM_NO_LIGHT_SLEEP

Prevent the system from going into light sleep. Argument is unused and should be set to 0.

2.9.26 POSIX Threads Support

Overview

ESP-IDF is based on FreeRTOS but offers a range of POSIX-compatible APIs that allow easy porting of third-party code. This includes support for common parts of the POSIX Threads pthread API.

POSIX Threads are implemented in ESP-IDF as wrappers around equivalent FreeRTOS features. The runtime memory or performance overhead of using the pthreads API is quite low, but not every feature available in either pthreads or FreeRTOS is available via the ESP-IDF pthreads support.

Pthreads can be used in ESP-IDF by including standard pthread.h header, which is included in the toolchain libc. An additional ESP-IDF specific header, esp_pthread.h, provides additional non-POSIX APIs for using some ESP-IDF features with pthreads.

C++ Standard Library implementations for std::thread, std::mutex, std::condition_variable, etc., are realized using pthreads (via GCC libstdc++). Therefore, restrictions mentioned here also apply to the equivalent C++ standard library functionality.

RTOS Integration

Unlike many operating systems using POSIX Threads, ESP-IDF is a real-time operating system with a real-time scheduler. This means that a thread will only stop running if a higher priority task is ready to run, the thread blocks on an OS synchronization structure like a mutex, or the thread calls any of the functions sleep, vTaskDelay(), or usleep.
Note: When calling a standard libc or C++ sleep function, such as `usleep` defined in `unistd.h`, the task will only block and yield the core if the sleep time is longer than one FreeRTOS tick period. If the time is shorter, the thread will busy-wait instead of yielding to another RTOS task.

By default, all POSIX Threads have the same RTOS priority, but it is possible to change this by calling a custom API.

Standard Features

The following standard APIs are implemented in ESP-IDF.

Refer to standard POSIX Threads documentation, or `pthread.h`, for details about the standard arguments and behaviour of each function. Differences or limitations compared to the standard APIs are noted below.

Thread APIs

- `pthread_create()`
  - The `attr` argument is supported for setting stack size and detach state only. Other attribute fields are ignored.
  - Unlike FreeRTOS task functions, the `start_routine` function is allowed to return. A detached type thread is automatically deleted if the function returns. The default joinable type thread will be suspended until `pthread_join()` is called on it.
- `pthread_join()`
- `pthread_detach()`
- `pthread_exit()`
- `sched_yield()`
- `pthread_self()`
  - An assert will fail if this function is called from a FreeRTOS task which is not a pthread.
- `pthread_equal()`

Thread Attributes

- `pthread_attr_init()`
- `pthread_attr_destroy()`
  - This function does not need to free any resources and instead resets the `attr` structure to defaults.
  - The implementation is the same as `pthread_attr_init()`.
- `pthread_attr_getstacksize()` / `pthread_attr_setstacksize()`
- `pthread_attr_getdetachstate()` / `pthread_attr_setdetachstate()`

Once

- `pthread_once()`

Static initializer constant `PTHREAD_ONCE_INIT` is supported.

Note: This function can be called from tasks created using either pthread or FreeRTOS APIs.

Mutexes

POSIX Mutexes are implemented as FreeRTOS Mutex Semaphores (normal type for “fast” or “error check” mutexes, and Recursive type for “recursive” mutexes). This means that they have the same priority inheritance behavior as mutexes created with `xSemaphoreCreateMutex()`.

- `pthread_mutex_init()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `pthread_mutex_timedlock()`
Chapter 2. API Reference

• pthread_mutex_trylock()
• pthread_mutex_unlock()
• pthread_mutexattr_init()
• pthread_mutexattr_destroy()
• pthread_mutexattr_gettype() / pthread_mutexattr_settype()

Static initializer constant `PTHREAD_MUTEX_INITIALIZER` is supported, but the non-standard static initializer constants for other mutex types are not supported.

**Note:** These functions can be called from tasks created using either pthread or FreeRTOS APIs.

Condition Variables

• `pthread_cond_init()`
  - The `attr` argument is not implemented and is ignored.
• `pthread_cond_destroy()`
• `pthread_cond_signal()`
• `pthread_cond_broadcast()`
• `pthread_cond_wait()`
• `pthread_cond_timedwait()`

Static initializer constant `PTHREAD_COND_INITIALIZER` is supported.

• The resolution of `pthread_cond_timedwait()` timeouts is the RTOS tick period (see `CONFIG_FREERTOS_HZ`). Timeouts may be delayed up to one tick period after the requested timeout.

**Note:** These functions can be called from tasks created using either pthread or FreeRTOS APIs.

Semaphores

In ESP-IDF, POSIX unnamed semaphores are implemented. The accessible API is described below. It implements semaphores as specified in the POSIX standard, unless specified otherwise.

• `sem_init()`
• `sem_destroy()`
  - `pshared` is ignored. Semaphores can always be shared between FreeRTOS tasks.
• `sem_post()`
  - If the semaphore has a value of `SEM_VALUE_MAX` already, -1 is returned and `errno` is set to EAGAIN.
• `sem_wait()`
• `sem_trywait()`
• `sem_timedwait()`
  - The time value passed by abstime will be rounded up to the next FreeRTOS tick.
  - The actual timeout happens after the tick that the time was rounded to and before the following tick.
  - It is possible, though unlikely, that the task is preempted directly after the timeout calculation, delaying the timeout of the following blocking operating system call by the duration of the preemption.
• `sem_getvalue()`

Read/Write Locks

The following API functions of the POSIX reader-writer locks specification are implemented:

• `pthread_rwlock_init()`
  - The `attr` argument is not implemented and is ignored.
• `pthread_rwlock_destroy()`
• `pthread_rwlock_rdlock()`
• `pthread_rwlock_tryrdlock()`
• `pthread_rwlock_wrlock()`
• `pthread_rwlock_trywrlock()`
• `pthread_rwlock_unlock()`
The static initializer constant PTHREAD_RWLOCK_INITIALIZER is supported.

**Note:** These functions can be called from tasks created using either pthread or FreeRTOS APIs.

### Thread-Specific Data

- **pthread_key_create()**
  - The `destr_function` argument is supported and will be called if a thread function exits normally, calls `pthread_exit()`, or if the underlying task is deleted directly using the FreeRTOS function `vTaskDelete()`.
  - `pthread_key_delete()`
  - `pthread_setspecific()` / `pthread_getspecific()`

**Note:** These functions can be called from tasks created using either pthread or FreeRTOS APIs. When calling these functions from tasks created using FreeRTOS APIs, `CONFIG_FREERTOS_TLSP_DELETION_CALLBACKS` config option must be enabled to ensure the thread-specific data is cleaned up before the task is deleted.

**Note:** There are other options for thread local storage in ESP-IDF, including options with higher performance. See *Thread Local Storage*.

### Not Implemented

The `pthread.h` header is a standard header and includes additional APIs and features which are not implemented in ESP-IDF. These include:

- `pthread_cancel()` returns ENOSYS if called.
- `pthread_condattr_init()` returns ENOSYS if called.

Other POSIX Threads functions (not listed here) are not implemented and will produce either a compiler or a linker error if referenced from an ESP-IDF application. If you identify a useful API that you would like to see implemented in ESP-IDF, please open a feature request on GitHub with the details.

### ESP-IDF Extensions

The API `esp_pthread_set_cfg()` defined in the `esp_pthreads.h` header offers custom extensions to control how subsequent calls to `pthread_create()` behaves. Currently, the following configuration can be set:

- Default stack size of new threads, if not specified when calling `pthread_create()` (overrides `CONFIG_PTHREAD_TASK_STACK_SIZE_DEFAULT`).
- RTOS priority of new threads (overrides `CONFIG_PTHREAD_TASK_PRIO_DEFAULT`).
- FreeRTOS task name for new threads (overrides `CONFIG_PTHREAD_TASK_NAME_DEFAULT`).

This configuration is scoped to the calling thread (or FreeRTOS task), meaning that `esp_pthread_set_cfg()` can be called independently in different threads or tasks. If the `inherit_cfg` flag is set in the current configuration then any new thread created will inherit the creator’s configuration (if that thread calls `pthread_create()` recursively), otherwise the new thread will have the default configuration.

### Examples

- `system/pthread` demonstrates using the pthreads API to create threads.
- `cxx/pthread` demonstrates using C++ Standard Library functions with threads.
API Reference

Header File

- components/pthread/include esp_pthread.h
- This header file can be included with:
  
```
#include "esp_pthread.h"
```
- This header file is a part of the API provided by the pthread component. To declare that your component depends on pthread, add the following to your CMakeLists.txt:
  
```
REQUIRES pthread
```
  
```
or
PRIV_REQUIRES pthread
```

Functions

```c
esp_pthread_cfg_t esp_pthread_get_default_config (void)
```

Creates a default pthread configuration based on the values set via menuconfig.

- Returns A default configuration structure.

```c
esp_err_t esp_pthread_set_cfg (const esp_pthread_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.

- **Parameters**
  - `cfg` - The pthread config parameters
- **Returns**
  - 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

```c
esp_err_t esp_pthread_get_cfg (esp_pthread_cfg_t *p)
```

Get current pthread creation configuration.

This will retrieve the current configuration that will be used for creating threads.

- **Parameters**
  - `p` - Pointer to the pthread config structure that will be updated with the currently configured parameters
- **Returns**
  - ESP_OK if the configuration was available
  - ESP_ERR_NOT_FOUND if a configuration wasn’t previously set

```c
esp_err_t esp_pthread_init (void)
```

Initialize pthread library.

**Structures**

```c
struct esp_pthread_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

2.9.27 Random Number Generation

ESP32-S2 contains a hardware random number generator, values from it can be obtained using the APIs esp_random() and esp_fill_random().

The hardware RNG produces true random numbers under any of the following conditions:

- RF subsystem is enabled (i.e., Wi-Fi is enabled).
- An internal entropy source has been enabled by calling bootloader_random_enable() and not yet disabled by calling bootloader_random_disable().
- While the ESP-IDF Second Stage Bootloader is running. This is because the default ESP-IDF bootloader implementation calls bootloader_random_enable() when the bootloader starts, and bootloader_random_disable() before executing the app.

When any of these conditions are true, samples of physical noise are continuously mixed into the internal hardware RNG state to provide entropy. Consult the ESP32-S2 Technical Reference Manual > Random Number Generator (RNG) [PDF] chapter for more details.

If none of the above conditions are true, the output of the RNG should be considered pseudo-random only.

Startup

During startup, ESP-IDF bootloader temporarily enables a non-RF entropy source (internal reference voltage noise) that provides entropy for any first boot key generation. However, after the app starts executing then normally only pseudo-random numbers are available until Wi-Fi is initialized.

To re-enable the entropy source temporarily during app startup, or for an application that does not use Wi-Fi, call the function bootloader_random_enable() to re-enable the internal entropy source. The function bootloader_random_disable() must be called to disable the entropy source again before using ADC, Wi-Fi.

Note: The entropy source enabled during the boot process by the ESP-IDF Second Stage Bootloader seeds the internal RNG state with some entropy. However, the internal hardware RNG state is not large enough to provide a
continuous stream of true random numbers. This is why a continuous entropy source must be enabled whenever true random numbers are required.

**Note:** If an application requires a source of true random numbers but it is not possible to permanently enable a hardware entropy source, consider using a strong software DRBG implementation such as the mbedTLS CTR-DRBG or HMAC-DRBG, with an initial seed of entropy from hardware RNG true random numbers.

**Secondary Entropy**

ESP32-S2 RNG contains a secondary entropy source, based on sampling an asynchronous 8 MHz internal oscillator (see the Technical Reference Manual for details). This entropy source is always enabled in ESP-IDF and continuously mixed into the RNG state by hardware. In testing, this secondary entropy source was sufficient to pass the Dieharder random number test suite without the main entropy source enabled (test input was created by concatenating short samples from a continuously resetting ESP32-S2). However, it is currently only guaranteed that true random numbers are produced when the main entropy source is also enabled as described above.

**API Reference**

**Header File**

- components/esp_hw_support/include/esp_random.h
- This header file can be included with:

```
#include "esp_random.h"
```

**Functions**

- `uint32_t esp_random (void)`
  - Get one random 32-bit word from hardware RNG.
  - If Wi-Fi or Bluetooth are enabled, this function returns true random numbers. In other situations, if true random numbers are required then consult the ESP-IDF Programming Guide “Random Number Generation” section for necessary prerequisites.
  - This function automatically busy-waits to ensure enough external entropy has been introduced into the hardware RNG state, before returning a new random number. This delay is very short (always less than 100 CPU cycles).

  **Returns** 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 is implemented via calls to esp_random(), so the same constraints apply.

**Parameters**

- `buf` – Pointer to buffer to fill with random numbers.
- `len` – Length of buffer in bytes

**Header File**

- components/bootloader_support/include/bootloader_random.h
- This header file can be included with:

```
#include "bootloader_random.h"
```
This header file is a part of the API provided by the bootloader_support component. To declare that your component depends on bootloader_support, add the following to your CMakeLists.txt:

```cmake
REQUIRES bootloader_support
```

or

```cmake
PRIV_REQUIRES bootloader_support
```

## Functions

**void bootloader_random_enable (void)**

Enable an entropy source for RNG if RF subsystem is disabled.

The exact internal entropy source mechanism depends on the chip in use but all SoCs use the SAR ADC to continuously mix random bits (an internal noise reading) into the HWRNG. Consult the SoC Technical Reference Manual for more information.

Can also be called from app code, if true random numbers are required without initialized RF subsystem. This might be the case in early startup code of the application when the RF subsystem has not started yet or if the RF subsystem should not be enabled for power saving.


**Warning:** This function is not safe to use if any other subsystem is accessing the RF subsystem or the ADC at the same time!

**void bootloader_random_disable (void)**

Disable entropy source for RNG.

Disables internal entropy source. Must be called after bootloader_random_enable() and before RF subsystem features, ADC, or I2S (ESP32 only) are initialized.

Consult the ESP-IDF Programming Guide “Random Number Generation” section for details.

**void bootloader_fill_random (void *buffer, size_t length)**

Fill buffer with ‘length’ random bytes.

**Note:** If this function is being called from app code only, and never from the bootloader, then it’s better to call esp_fill_random().

### Parameters

- `buffer` - Pointer to buffer
- `length` - This many bytes of random data will be copied to buffer

**getrandom()**

A compatible version of the Linux getrandom() function is also provided for ease of porting:

```c
#include <sys/random.h>

ssize_t getrandom(void *buf, size_t buflen, unsigned int flags);
```

This function is implemented by calling esp_fill_random() internally.

The `flags` argument is ignored, this function is always non-blocking but the strength of any random numbers is dependent on the same conditions described above.

Return value is -1 (with errno set to EFAULT) if the `buf` argument is NULL, and equal to `buflen` otherwise.
**getentropy()**

A compatible version of the Linux `getentropy()` function is also provided for ease of porting:

```c
#include <unistd.h>

int getentropy(void *buffer, size_t length);
```

This function is implemented by calling `getrandom()` internally.

Strength of any random numbers is dependent on the same conditions described above.

**Return value is 0 on success and -1 otherwise with errno set to:**

- EFAULT if the `buffer` argument is NULL.
- EIO if the `length` is more then 256.

### 2.9.28 Sleep Modes

#### Overview

ESP32-S2 supports two major power saving modes: Light-sleep and Deep-sleep. According to the features used by an application, there are some sub sleep modes. See **Sleep Modes** for these sleep modes and sub sleep modes. Additionally, there are some power-down options that can be configured to further reduce the power consumption. See **Power-down Options** for more details.

There are several wakeup sources in the sleep modes. These sources can also be combined so that the chip will wake up when any of the sources are triggered. **Wakeup Sources** describes these wakeup sources and configuration APIs in detail.

The configuration of power-down options and wakeup sources are optional. They can be configured at any moment before entering the sleep modes.

Then the application can call sleep start APIs to enter one of the sleep modes. See **Entering Sleep** for more details. When the wakeup condition is met, the application is awoken from sleep. See **Checking Sleep Wakeup Cause** on how to get the wakeup cause, and **Disable Sleep Wakeup Source** on how to handle the wakeup sources after wakeup.

#### Sleep Modes

In Light-sleep mode, the digital peripherals, most of the RAM, and CPUs are clock-gated and their supply voltage is reduced. Upon exit from Light-sleep, the digital peripherals, RAM, and CPUs resume operation and their internal states are preserved.

In Deep-sleep mode, the CPUs, most of the RAM, and all digital peripherals that are clocked from APB_CLK are powered off. The only parts of the chip that remain powered on are:

- RTC controller
- ULP coprocessor
- RTC FAST memory
- RTC SLOW memory

**Wi-Fi and Sleep Modes** In Deep-sleep and Light-sleep modes, the wireless peripherals are powered down. Before entering Deep-sleep or Light-sleep modes, applications must disable Wi-Fi using the appropriate calls (`esp_wifi_stop()`). Wi-Fi connections are not maintained in Deep-sleep or Light-sleep mode, even if these functions are not called.
If Wi-Fi connections need to be maintained, enable Wi-Fi Modem-sleep mode and automatic Light-sleep feature (see Power Management APIs). This will allow the system to wake up from sleep automatically when required by the Wi-Fi driver, thereby maintaining a connection to the AP.

**Sub Sleep Modes** Tables below list the sub sleep modes in the first row and the features they support in the first column. Modes that support more features may consume more power during sleep mode. The sleep system automatically selects the mode that satisfies all the features required by the user while consuming least power.

### Deep-sleep

<table>
<thead>
<tr>
<th>Sub Sleep Mode</th>
<th>DSLP_ULTRA_LOW</th>
<th>DSLP_DEFAULT</th>
<th>DSLP_8MD256/DSLP_ADC_TSENS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULP/Touch sensor (ESP32-S2 and ESP32-S3 only)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>RTC IO input/RTC memory at high temperature</td>
<td></td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ADC_TSEN_MONITOR</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>8MD256 as the clock source for RTC_SLOW_CLK</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

**Features:**

1. **RTC IO input/RTC memory at high temperature (experimental):** Use RTC IO as input pins, or use RTC memory at high temperature. The chip can go into ultra low power mode when these features are disabled. Controlled by API `rtc_sleep_enable_ultra_low()`.

2. **ADC_TSEN_MONITOR:** Use ADC/Temperature Sensor in monitor mode (controlled by ULP). Enabled by API `ulp_adc_init()` or its higher level APIs. Only available for ESP32-S2 and ESP32-S3 chips with monitor mode.

3. **8MD256 as the clock source for RTC_SLOW_CLK:** When 8MD256 is selected as the clock source for RTC_SLOW_CLK using the Kconfig option `CONFIG_RTC_CLK_SRC_INT_8MD256`, the chip will automatically enter this sub sleep mode during Deep-sleep mode.

### Light-sleep

<table>
<thead>
<tr>
<th>Sub Sleep Mode</th>
<th>LSLP_DEFAULT</th>
<th>LSLP_ADC_TSENS</th>
<th>LSLP_8MD256</th>
<th>LSLP_LEDC8M</th>
<th>LSLP_XTAL_FPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULP/Touch sensor (ESP32-S2 and ESP32-S3 only)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>RTC IO input/RTC memory at high temperature</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ADC_TSEN_MONITOR</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8MD256 as the clock source for RTC_SLOW_CLK</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 MHz RC clock source used by digital peripherals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Keep the XTAL clock on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

**Features:** (Also see 8MD256 and ADC_TSEN_MONITOR features for Deep-sleep mode above)

1. **8 MHz RC clock source used by digital peripherals:** Currently, only LEDC uses this clock source during Light-sleep mode. When LEDC selects this clock source, this feature is automatically enabled.

2. **Keep the XTAL clock on:** Keep the XTAL clock on during Light-sleep mode. Controlled by `ESP_PD_DOMAIN_XTAL` power domain.

ESP32-S2 uses the same power mode for LSLP_8MD256, LSLP_LEDC8M, and LSLP_XTAL_FPU features.
Chapter 2. API Reference

Wakeup Sources

Wakeup sources can be enabled using `esp_sleep_enable_X_wakeup` APIs. Wakeup sources are not disabled after wakeup, you can disable them using `esp_sleep_disable_wakeup_source()` API if you do not need them any more. See Disable Sleep Wakeup Source.

Following are the wakeup sources supported on ESP32-S2.

**Timer**  The 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.

For details on RTC clock options, see ESP32-S2 Technical Reference Manual > ULP Coprocessor [PDF].

RTC peripherals or RTC memories do not need to be powered on during sleep in this wakeup mode.

`esp_sleep_enable_timer_wakeup()` function can be used to enable sleep wakeup using a timer.

**Touchpad**  The RTC IO module contains the logic to trigger wakeup when a touch sensor interrupt occurs. To wakeup from a touch sensor interrupt, users need to configure the touch pad interrupt before the chip enters Deep-sleep or Light-sleep modes.

`esp_sleep_enable_touchpad_wakeup()` function can be used to enable this wakeup source.

**External Wakeup (ext0)**  The RTC IO module contains the logic to trigger wakeup when one of RTC GPIOs is set to a predefined logic level. RTC IO is part of the RTC peripherals power domain, so RTC peripherals will be kept powered on during Deep-sleep if this wakeup source is requested.

The RTC IO module is enabled in this mode, so 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_deep_sleep_start()`.

`esp_sleep_enable_ext0_wakeup()` function can be used to enable this wakeup source.

| Warning: | After waking up from sleep, the IO pad used for wakeup will be configured as RTC IO. Therefore, before using this pad as digital GPIO, users need to reconfigure it using `rtc_gpio_deinit()` function. |

**External Wakeup (ext1)**  The RTC controller contains the logic to trigger wakeup using multiple RTC GPIOs.

One of the following two logic functions can be used to trigger general ext1 wakeup:

- **wake up if any of the selected pins is high (ESP_EXT1_WAKEUP_ANY_HIGH)**
- **wake up if any of the selected pins is low (ESP_EXT1_WAKEUP_ANY_LOW)**

This wakeup source is controlled by the RTC controller. Unlike ext0, this wakeup source supports wakeup even when the RTC peripheral is powered down. Although the power domain of the RTC peripheral, where RTC IOs are located, is powered down during sleep modes, ESP-IDF will automatically lock the state of the wakeup pin before the system enters sleep modes and unlock upon exiting sleep modes. Therefore, the internal pull-up or pull-down resistors can still be configured for the wakeup pin:

```c
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_ON);
rtc_gpio_pullup_dis(gpio_num);
rtc_gpio_pulldown_en(gpio_num);
```

If we turn off the RTC_PERIPH domain, we will use the HOLD feature to maintain the pull-up and pull-down on the pins during sleep. HOLD feature will be acted on the pin internally before the system enters sleep modes, and this can further reduce power consumption:

```c
rtc_gpio_pullup_dis(gpio_num);
rtc_gpio_pulldown_en(gpio_num);
```
If certain chips lack the RTC_PERIPH domain, we can only use the HOLD feature to maintain the pull-up and pull-down on the pins during sleep modes:

```c
gpio_pullup_dis(gpio_num);
gpio_pulldown_en(gpio_num);
```

`esp_sleep_enable_ext1_wakeup()` function can be used to enable this wakeup source for general ext1 wakeups.

**Warning:**
- To use the EXT1 wakeup, the IO pad(s) are configured as RTC IO. Therefore, before using these pads as digital GPIOs, users need to reconfigure them by calling the `rtc_gpio_deinit()` function.
- If the RTC peripherals are configured to be powered down (which is by default), the wakeup IOs will be set to the holding state before entering sleep. Therefore, after the chip wakes up from Light-sleep, please call `rtc_gpio_hold_dis` to disable the hold function to perform any pin re-configuration. For Deep-sleep wakeup, this is already being handled at the application startup stage.

### 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 the 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.

`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.

**Warning:** Before entering Light-sleep mode, check if any GPIO pin to be driven is part of the VDD_SPI power domain. If so, this power domain must be configured to remain ON during sleep.

For example, on ESP32-WROOM-32 board, GPIO16 and GPIO17 are linked to VDD_SPI power domain. If they are configured to remain high during Light-sleep, the power domain should be configured to remain powered ON. This can be done with `esp_sleep_pd_config()`:

```c
esp_sleep_pd_config(ESP_PD_DOMAIN_VDDSDIO, ESP_PD_OPTION_ON);
```

### UART Wakeup (Light-sleep Only)
When ESP32-S2 receives UART input from external devices, it is often necessary to wake up the chip when input data is available. The 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-S2 to trigger wakeup before sending the data.

`esp_sleep_enable_uart_wakeup()` function can be used to enable this wakeup source.

### Disable Sleep Wakeup Source
Previously configured wakeup sources 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.
Power-down Options

The application can force specific powerdown modes for RTC peripherals and RTC memories. In Deep-sleep mode, we can also isolate some IOs to further reduce current consumption.

**Power-down of RTC Peripherals and Memories** By default, `esp_deep_sleep_start()` and `esp_light_sleep_start()` functions power down all RTC power domains which are not needed by the enabled wakeupsources. To override this behaviour, `esp_sleep_pd_config()` function is provided.

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.

**Power-down of Flash** By default, to avoid potential issues, `esp_light_sleep_start()` function does **not** power down flash. To be more specific, it takes time to power down the flash and during this period the system may be woken up, which then actually powers up the flash before this flash could be powered down completely. As a result, there is a chance that the flash may not work properly.

So, in theory, it is ok if you only wake up the system after the flash is completely powered down. However, in reality, the flash power-down period can be hard to predict (for example, this period can be much longer when you add filter capacitors to the flash’s power supply circuit) and uncontrollable (for example, the asynchronous wake-up signals make the actual sleep time uncontrollable).

**Warning:** If a filter capacitor is added to your flash power supply circuit, please do everything possible to avoid powering down flash.

Therefore, it is recommended not to power down flash when using ESP-IDF. For power-sensitive applications, it is recommended to use Kconfig option `CONFIG_ESP_SLEEP_FLASH_LEAKAGE_WORKAROUND` to reduce the power consumption of the flash during Light-sleep, instead of powering down the flash.

It is worth mentioning that PSRAM has a similar Kconfig option `CONFIG_ESP_SLEEP_PSRAM_LEAKAGE_WORKAROUND`.

However, for those who have fully understood the risk and are still willing to power down the flash to further reduce the power consumption, please check the following mechanisms:

- Setting Kconfig option `CONFIG_ESP_SLEEP_POWER_DOWN_FLASH` only powers down the flash when the RTC timer is the only wake-up source **and** the sleep time is longer than the flash power-down period.
- Calling `esp_sleep_pd_config(ESP_PD_DOMAIN_VDDSIO, ESP_PD_OPTION_OFF)` powers down flash when the RTC timer is not enabled as a wake-up source **or** the sleep time is longer than the flash power-down period.

**Note:**

- ESP-IDF does not provide any mechanism that can power down the flash in all conditions when Light-sleep.
- `esp_deep_sleep_start()` function forces power down flash regardless of user configuration.

**Configuring IOs (Deep-sleep Only)** Some ESP32-S2 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 to prevent extra current draw, call `rtc_gpio_isolate()` function.
For example, on ESP32-WROVER module, GPIO12 is pulled up externally, and it also has an internal pulldown in the ESP32 chip. This means that in Deep-sleep, some current flows 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 such extra current:

```c
rtc_gpio_isolate(GPIO_NUM_12);
```

### Entering Sleep

`esp_light_sleep_start()` or `esp_deep_sleep_start()` functions can be used to enter Light-sleep or Deep-sleep modes correspondingly. After that, the system configures the parameters of RTC controller according to the requested wakeup sources and power-down options.

It is also possible to enter sleep modes with no wakeup sources configured. In this case, the chip will be in sleep modes indefinitely until external reset is applied.

**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 touchpad, it is possible to identify which touch pin has caused wakeup using `esp_sleep_get_touchpad_wakeup_status()` functions.

For ext1 wakeup sources, it is possible to identify which touch pin has caused wakeup using `esp_sleep_get_ext1_wakeup_status()` functions.

### Application Example

- **protocols/sntp**: the implementation of basic functionality of Deep-sleep, where ESP module is periodically waken up to retrieve time from NTP server.
- **wifi/power_save**: the usage of Wi-Fi Modem-sleep mode and automatic Light-sleep feature to maintain Wi-Fi connections.
- **system/deep_sleep**: the usage of various Deep-sleep wakeup triggers and ULP coprocessor programming.

### API Reference

#### Header File

- **components/esp_hw_support/include/esp_sleep.h**
- This header file can be included with:

  ```c
  #include "esp_sleep.h"
  ```
Chapter 2. API Reference

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_deep_sleep_start/esp_light_sleep_start function.

**Parameters**

- `source` -- number of source to disable of type esp_sleep_source_t

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_STATE if trigger was not active

`esp_err_t esp_sleep_enable_ulp_wakeup(void)`

Enable wakeup by ULP coprocessor.

**Note:** On ESP32, 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.

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_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.

**Parameters**

- `time_in_us` -- time before wakeup, in microseconds

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_ARG if value is out of range (TBD)

`esp_err_t esp_sleep_enable_touchpad_wakeup(void)`

Enable wakeup by touch sensor.

**Note:** On ESP32, touch wakeup source can not 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.

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if additional current by touch (CONFIG_RTC_EXT_CRYST_ADDIT_CURRENT) is enabled.
- ESP_ERR_INVALID_STATE if wakeup triggers conflict
The text contains API references for ESP32, including functions for handling touchpad and GPIOs. Here is a breakdown of the key points:

- `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;
  - **Returns** touch pad which caused wakeup

- `bool esp_sleep_is_valid_wakeup_gpio(gpio_num_t gpio_num)`
  - Returns true if a GPIO number is valid for use as a wakeup source.
  - **Note:** For SoCs with RTC IO capability, this can be any valid RTC IO input pin.
  - **Parameters**
    - `gpio_num` – Number of the GPIO to test for wakeup source capability
  - **Returns**
    - True if this GPIO number will be accepted as a sleep wakeup source.

- `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_deep_sleep_start/esp_light_sleep_start, immediately before entering sleep mode.
  - **Parameters**
    - `gpio_num` – GPIO number used as wakeup source. Only GPIOs with the RTC functionality can be used. For different SoCs, the related GPIOs are:
      - ESP32: 0, 2, 4, 12-15, 25-27, 32-39;
      - ESP32-S2: 0-21;
      - ESP32-S3: 0-21.
    - `level` – input level which will trigger wakeup (0=low, 1=high)
  - **Returns**
    - 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

- `esp_err_t esp_sleep_enable_ext1_wakeup(uint64_t io_mask, esp_sleep_ext1_wakeup_mode_t level_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 selected pins go into the state given by level_mode argument, the chip will be woken up.
  - **Note:** This function does not modify pin configuration. The pins are configured in esp_deep_sleep_start/esp_light_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. If we turn off the RTC_PERIPH domain or certain chips lack the RTC_PERIPH domain, we will use the HOLD feature to maintain the pull-up and pull-down on the pins during sleep. HOLD feature will be acted on the pin internally before the system entering sleep, and this can further reduce power consumption.

**Parameters**

- **io_mask** – Bit mask of GPIO numbers which will cause wakeup. Only GPIOs which have RTC functionality can be used in this bit map. For different SoCs, the related GPIOs are:
  - ESP32: 0, 2, 4, 12-15, 25-27, 32-39
  - ESP32-S2: 0-21
  - ESP32-S3: 0-21
  - ESP32-C6: 0-7
  - ESP32-H2: 7-14

- **level_mode** – Select logic function used to determine wakeup condition: When target chip is ESP32:
  - 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

When target chip is ESP32-S2, ESP32-S3, ESP32-C6 or ESP32-H2:

- ESP_EXT1_WAKEUP_ANY_LOW: wake up when any of the selected GPIOs is low
- ESP_EXT1_WAKEUP_ANY_HIGH: wake up when any of the selected GPIOs is high

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_ARG if any of the selected GPIOs is not an RTC GPIO, or mode is invalid

```
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 wake up 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:** On ESP32, GPIO wakeup source can not be used together with touch or ULP wakeup sources.

**Returns**

- 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.

**Parameters**

- **uart_num** – UART port to wake up from
**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_ARG if wakeup from given UART is not supported

`esp_err_t esp_sleep_enable_bt_wakeup (void)`

Enable wakeup by bluetooth.

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if wakeup from bluetooth is not supported

`esp_err_t esp_sleep_disable_bt_wakeup (void)`

Disable wakeup by bluetooth.

**Returns**

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if wakeup from bluetooth is not supported

`esp_err_t esp_sleep_enable_wifi_wakeup (void)`

Enable wakeup by WiFi MAC.

**Returns**

- ESP_OK on success

`esp_err_t esp_sleep_disable_wifi_wakeup (void)`

Disable wakeup by WiFi MAC.

**Returns**

- ESP_OK on success

`esp_err_t esp_sleep_enable_wifi_beacon_wakeup (void)`

Enable beacon wakeup by WiFi MAC, it will wake up the system into modem state.

**Returns**

- ESP_OK on success

`esp_err_t esp_sleep_disable_wifi_beacon_wakeup (void)`

Disable beacon wakeup by WiFi MAC.

**Returns**

- ESP_OK on success

`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.

**Returns**

- bit mask, if GPIO on 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 set using this API, all power domains default to ESP_PD_OPTION_AUTO.

**Parameters**

- `domain` - power domain to configure
- `option` - power down option (ESP_PD_OPTION_OFF, ESP_PD_OPTION_ON, or ESP_PD_OPTION_AUTO)

**Returns**

- ESP_OK on success
- ESP_ERR_INVALID_ARG if either of the arguments is out of range
**esp_err_t esp_deep_sleep_try_to_start (void)**

Enter deep sleep with the configured wakeup options.

The reason for the rejection can be such as a short sleep time.

**Note:** In general, the function does not return, but if the sleep is rejected, then it returns from it.

**Returns**
- No return - If the sleep is not rejected.
- ESP_ERR_SLEEP_REJECT sleep request is rejected(wakeup source set before the sleep request)

**void esp_deep_sleep_start (void)**

Enter deep sleep with the configured wakeup options.

**Note:** The function does not do a return (no rejection). Even if wakeup source set before the sleep request it goes to deep sleep anyway.

**esp_err_t esp_light_sleep_start (void)**

Enter light sleep with the configured wakeup options.

**Returns**
- ESP_OK on success (returned after wakeup)
- ESP_ERR_SLEEP_REJECT sleep request is rejected(wakeup source set before the sleep request)
- ESP_ERR_SLEEP_TOO_SHORT_SLEEP_DURATION after deducting the sleep flow overhead, the final sleep duration is too short to cover the minimum sleep duration of the chip, when rtc timer wakeup source enabled

**esp_err_t esp_deep_sleep_try (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.

The reason for the rejection can be such as a short sleep time.

**Note:** In general, the function does not return, but if the sleep is rejected, then it returns from it.

**Parameters**
- **time_in_us** - deep-sleep time, unit: microsecond

**Returns**
- No return - If the sleep is not rejected.
- ESP_ERR_SLEEP_REJECT sleep request is rejected(wakeup source set before the sleep request)

**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.

**Note:** The function does not do a return (no rejection). Even if wakeup source set before the sleep request it goes to deep sleep anyway.

### Parameters

**time** - deep-sleep time, unit: microsecond

**esp_err_t esp_deep_sleep_register_hook** *(esp_deep_sleep_cb_t new_dslp_cb)*

Register a callback to be called from the deep sleep prepare.

**Warning:** deepsleep callbacks should without parameters, and MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

### Parameters

**new_dslp_cb** - Callback to be called

### Returns

- ESP_OK: Callback registered to the deepsleep misc_modules_sleep_prepare
- ESP_ERR_NO_MEM: No more hook space for register the callback

**void esp_deep_sleep_deregister_hook** *(esp_deep_sleep_cb_t old_dslp_cb)*

Unregister an deepsleep callback.

### Parameters

**old_dslp_cb** - Callback to be unregistered

**esp_sleep_wakeup_cause_t esp_sleep_get_wakeup_cause** *(void)*

Get the wakeup source which caused wakeup from sleep.

### Returns

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().

**void esp_set_deep_sleep_wake_stub_default_entry** *(void)*

Set wake stub entry to default esp_wake_stub_entry

**esp_deep_sleep_wake_stub_fn_t esp_get_deep_sleep_wake_stub** *(void)*

Get current wake from deep sleep stub.

### Returns

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.
void esp_sleep_config_gpio_isolate (void)
    Configure to isolate all GPIO pins in sleep state.
void esp_sleep_enable_gpio_switch (bool enable)
    Enable or disable GPIO pins status switching between slept status and waked status.
    Parameters enable – decide whether to switch status or not

Macros

ESP_PD_DOMAIN_RTC8M

Type Definitions

typedef void (*esp_deep_sleep_cb_t)(void)
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:

    enumerator ESP_EXT1_WAKEUP_ANY_LOW
        Wake the chip when any of the selected GPIOs go low.

    enumerator ESP_EXT1_WAKEUP_ANY_HIGH
        Wake the chip when any of the selected GPIOs go high.

    enumerator ESP_EXT1_WAKEUP_ALL_LOW

enum esp_sleep_pd_domain_t
    Power domains which can be powered down in sleep mode.
    Values:

    enumerator ESP_PD_DOMAIN_RTC_PERIPH
        RTC IO, sensors and ULP co-processor.

    enumerator ESP_PD_DOMAIN_RTC_SLOW_MEM
        RTC slow memory.

    enumerator ESP_PD_DOMAIN_RTC_FAST_MEM
        RTC fast memory.
enumerator **ESP_PD_DOMAIN_XTAL**
    XTAL oscillator.

denumerator **ESP_PD_DOMAIN_RC_FAST**
    Internal Fast oscillator.

denumerator **ESP_PD_DOMAIN_VDDSDIO**
    VDD_SDIO.

denumerator **ESP_PD_DOMAIN_MAX**
    Number of domains.

denum **esp_sleep_pd_option_t**
    Power down options.
    
    **Values:**
    
    enumerator **ESP_PD_OPTION_OFF**
        Power down the power domain in sleep mode.
    
    enumerator **ESP_PD_OPTION_ON**
        Keep power domain enabled during sleep mode.
    
    enumerator **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:**
    
    enumerator **ESP_SLEEP_WAKEUP_UNDEFINED**
        In case of deep sleep, reset was not caused by exit from deep sleep.
    
    enumerator **ESP_SLEEP_WAKEUP_ALL**
        Not a wakeup cause, used to disable all wakeup sources with esp_sleep_disable_wakeup_source.
    
    enumerator **ESP_SLEEP_WAKEUP_EXT0**
        Wakeup caused by external signal using RTC_IO.
    
    enumerator **ESP_SLEEP_WAKEUP_EXT1**
        Wakeup caused by external signal using RTC_CNTL.
    
    enumerator **ESP_SLEEP_WAKEUP_TIMER**
        Wakeup caused by timer.
    
    enumerator **ESP_SLEEP_WAKEUP_TOUCHPAD**
        Wakeup caused by touchpad.
enumerator **ESP_SLEEP_WAKEUP_ULP**
Wakeup caused by ULP program.

enumerator **ESP_SLEEP_WAKEUP_GPIO**
Wakeup caused by GPIO (light sleep only on ESP32, S2 and S3)

enumerator **ESP_SLEEP_WAKEUP_UART**
Wakeup caused by UART (light sleep only)

enumerator **ESP_SLEEP_WAKEUP_WIFI**
Wakeup caused by WIFI (light sleep only)

enumerator **ESP_SLEEP_WAKEUP_COCPU**
Wakeup caused by COCPU int.

enumerator **ESP_SLEEP_WAKEUP_COCPU_TRAP_TRIG**
Wakeup caused by COCPU crash.

enumerator **ESP_SLEEP_WAKEUP_BT**
Wakeup caused by BT (light sleep only)

**enum esp_sleep_mode_t**
Sleep mode.

**Values:**

enumerator **ESP_SLEEP_MODE_LIGHT_SLEEP**
light sleep mode

enumerator **ESP_SLEEP_MODE_DEEP_SLEEP**
depth sleep mode

**enum [anonymous]**

**Values:**

enumerator **ESP_ERR_SLEEP_REJECT**

enumerator **ESP_ERR_SLEEP_TOO_SHORT_SLEEP_DURATION**

### 2.9.29 SoC Capabilities

This section lists the macro definitions of the ESP32-S2’s SoC hardware capabilities. These macros are commonly used by conditional-compilation directives (e.g., `#if`) in ESP-IDF to determine which hardware-dependent features are supported, thus control what portions of code are compiled.

**Warning:** These macro definitions are currently not considered to be part of the public API, and may be changed in a breaking manner (see ESP-IDF Versions for more details).
API Reference

Header File

- components/soc/esp32s2/include/soc/soc_caps.h
- This header file can be included with:
  ```c
  #include "soc/soc_caps.h"
  ```

Macros

- `SOC_ADC_SUPPORTED`
- `SOC_DAC_SUPPORTED`
- `SOC_UART_SUPPORTED`
- `SOC_TWAI_SUPPORTED`
- `SOC_CP_DMA_SUPPORTED`
- `SOC_DEDICATED_GPIO_SUPPORTED`
- `SOC_GPTIMER_SUPPORTED`
- `SOC_SUPPORTS_SECURE_DL_MODE`
- `SOC_ULP_FSM_SUPPORTED`
- `SOC_RISCV_COPROC_SUPPORTED`
- `SOC_USB_OTG_SUPPORTED`
- `SOC_PCNT_SUPPORTED`
- `SOC_WIFI_SUPPORTED`
- `SOC_CCOMP_TIMER_SUPPORTED`
- `SOC_ASYNC_MEMCPY_SUPPORTED`
- `SOC_EFUSE_KEY_PURPOSE_FIELD`
- `SOC_EFUSE_SUPPORTED`
- `SOC_TEMP_SENSOR_SUPPORTED`
SOC_CACHE_SUPPORT_WRAP
SOC_RTC_FAST_MEM_SUPPORTED
SOC_RTC_SLOW_MEM_SUPPORTED
SOC_RTC_MEM_SUPPORTED
SOC_PSRAM_DMA_CAPABLE
SOC_XT_WDT_SUPPORTED
SOC_I2S_SUPPORTED
SOC_RMT_SUPPORTED
SOC_SDM_SUPPORTED
SOC_GSPI_SUPPORTED
SOC_LEDC_SUPPORTED
SOC_I2C_SUPPORTED
SOC_SYSTIMER_SUPPORTED
SOC_SUPPORT_COEXISTENCE
SOC_AES_SUPPORTED
SOC_MPI_SUPPORTED
SOC_SHA_SUPPORTED
SOC_HMAC_SUPPORTED
SOC_DIG_SIGN_SUPPORTED
SOC_FLASH_ENC_SUPPORTED
SOC_SECURE_BOOT_SUPPORTED
SOC_MEMPROT_SUPPORTED
SOC_TOUCH_SENSOR_SUPPORTED
SOC_BOD_SUPPORTED
SOC_CLK_TREE_SUPPORTED
SOC_MPU_SUPPORTED
SOC_WDT_SUPPORTED
SOC_SPI_FLASH_SUPPORTED
SOC_XTAL_SUPPORT_40M
SOC_ADC_RTC_CTRL_SUPPORTED
<SAR ADC Module
SOC_ADC_DIG_CTRL_SUPPORTED
SOC_ADC_ARBITER_SUPPORTED
SOC_ADC_DIG_IIR_FILTER_SUPPORTED
SOC_ADC_DIG_IIR_FILTER_UNIT_BINDED
SOC_ADC_MONITOR_SUPPORTED
SOC_ADC_DMA_SUPPORTED
SOC_ADC_DIG_SUPPORTED_UNIT (UNIT)
SOC_ADC_PERIPH_NUM
SOC_ADC_CHANNEL_NUM (UNIT)
SOC_ADC_MAX_CHANNEL_NUM
SOC_ADC_ATTEN_NUM
Digital
SOC_ADC_DIGI_CONTROLLER_NUM
SOC_ADC_PATT_LEN_MAX
Two pattern table, each contains 16 items. Each item takes 1 byte
SOC_ADC_DIGI_MIN_BITWIDTH
SOC_ADC_DIGI_MAX_BITWIDTH
SOC_ADC_DIGI_IIR_FILTER_NUM

SOC_ADC_DIGI_RESULT_BYTES

SOC_ADC_DIGI_DATA_BYTES_PER_CONV

SOC_ADC_DIGI_MONITOR_NUM
  F_sample = F_digi_con / 2 / interval. F_digi_con = 5M for now. 30 <= interval <= 4095

SOC_ADC_SAMPLE_FREQ_THRES_HIGH

SOC_ADC_SAMPLE_FREQ_THRES_LOW
  RTC

SOC_ADC_RTC_MIN_BITWIDTH

SOC_ADC_RTC_MAX_BITWIDTH
  Calibration

SOC_ADC_CALIBRATION_V1_SUPPORTED
  support HW offset calibration version 1

SOC_ADC_SELF_HW_CALI_SUPPORTED
  support HW offset self calibration ADC power control is shared by PWDET, TempSensor

SOC_ADC_SHARED_POWER

SOC_BROWNOUT_RESET_SUPPORTED

SOC_CACHE_WRITEBACK_SUPPORTED

SOC_CP_DMA_MAX_BUFFER_SIZE
  Maximum size of the buffer that can be attached to descriptor

SOC_CPU_CORES_NUM

SOC_CPU_INTR_NUM

SOC_CPU_BREAKPOINTS_NUM

SOC_CPU_WATCHPOINTS_NUM

SOC_CPU_WATCHPOINT_MAX_REGION_SIZE

SOC_DAC_CHAN_NUM
SOC_DAC_RESOLUTION
SOC_GPIO_PORT
SOC_GPIO_PIN_COUNT
SOC_GPIO_SUPPORT_PIN_GLITCH_FILTER
SOC_GPIO_FILTER_CLK_SUPPORT_APB
SOC_GPIO_SUPPORT_RTC_INDEPENDENT
SOC_GPIO_SUPPORT_FORCE_HOLD
SOC_GPIO_VALID_GPIO_MASK
SOC_GPIO_VALID_OUTPUT_GPIO_MASK
SOC_GPIO_IN_RANGE_MAX
SOC_GPIO_OUT_RANGE_MAX
SOC_GPIO_VALID_DIGITAL_IO_PAD_MASK
SOC_GPIO_CLOCKOUT_BY_IO_MUX

SOC_DEDIC_GPIO_OUT_CHANNELS_NUM
  8 outward channels on each CPU core

SOC_DEDIC_GPIO_IN_CHANNELS_NUM
  8 inward channels on each CPU core

SOC_DEDIC_GPIO_ALLOW_REG_ACCESS
  Allow access dedicated GPIO channel by register

SOC_DEDIC_GPIO_HAS_INTERRUPT
  Dedicated GPIO has its own interrupt source

SOC_DEDIC_GPIO_OUT_AUTO_ENABLE
  Dedicated GPIO output attribution is enabled automatically

SOC_I2C_NUM

SOC_I2C_FIFO_LEN
  I2C hardware FIFO depth
Chapter 2. API Reference

**SOC_I2C_CMD_REG_NUM**
Number of I2C command registers

**SOC_I2C_SUPPORT_SLAVE**

**SOC_I2C_SUPPORT_HW_CLR_BUS**

**SOC_I2C_SUPPORT_REF_TICK**

**SOC_I2C_SUPPORT_APB**

**SOC_I2S_NUM**

**SOC_I2S_HW_VERSION_1**

**SOC_I2S_SUPPORTS_APLL**

**SOC_I2S_SUPPORTS_PLL_F160M**

**SOC_I2S_SUPPORTS_DMA_EQUAL**

**SOC_I2S_SUPPORTS_LCD_CAMERA**

**SOC_I2S_APLL_MIN_FREQ**

**SOC_I2S_APLL_MAX_FREQ**

**SOC_I2S_APLL_MIN_RATE**

**SOC_I2S_LCD_I80_VARIANT**

**SOC_LCD_I80_SUPPORTED**
Intel 8080 LCD is supported

**SOC_LCD_I80_BUSES**
Only I2S0 has LCD mode

**SOC_LCD_I80_BUS_WIDTH**
Intel 8080 bus width

**SOC_LEDC_HAS_TIMER_SPECIFIC_MUX**

**SOC_LEDC_SUPPORT_APB_CLOCK**

**SOC_LEDC_SUPPORT_REF_TICK**
SOC_LEDC_SUPPORT_XTAL_CLOCK
SOC_LEDC_CHANNEL_NUM
SOC_LEDC_TIMER_BIT_WIDTH
SOC_LEDC_SUPPORT_FADE_STOP
SOC_MMU_LINEAR_ADDRESS_REGION_NUM
SOC_MMU_PERIPH_NUM
SOC_MPU_CONFIGURABLE_REGIONS_SUPPORTED
SOC_MPU_MIN_REGION_SIZE
SOC_MPU_REGIONS_MAX_NUM
SOC_MPU_REGION_RO_SUPPORTED
SOC_MPU_REGION_WO_SUPPORTED
SOC_PCNT_GROUPS
SOC_PCNT_UNITS_PER_GROUP
SOC_PCNT_CHANNELS_PER_UNIT
SOC_PCNT_THRES_POINT_PER_UNIT
SOC_RMT_GROUPS
   One RMT group
SOC_RMT_TX_CANDIDATES_PER_GROUP
   Number of channels that capable of Transmit in each group
SOC_RMT_RX_CANDIDATES_PER_GROUP
   Number of channels that capable of Receive in each group
SOC_RMT_CHANNELS_PER_GROUP
   Total 4 channels
SOC_RMT_MEM_WORDS_PER_CHANNEL
   Each channel owns 64 words memory (1 word = 4 Bytes)
SOC_RMT_SUPPORT_RX_DEMODULATION
   Support signal demodulation on RX path (i.e. remove carrier)
SOC_RMT_SUPPORT_TX_ASYNC_STOP
Support stop transmission asynchronously

SOC_RMT_SUPPORT_TX_LOOP_COUNT
Support transmitting specified number of cycles in loop mode

SOC_RMT_SUPPORT_TX_SYNCHRO
Support coordinate a group of TX channels to start simultaneously

SOC_RMT_SUPPORT_TX_CARRIER_DATA_ONLY
TX carrier can be modulated to data phase only

SOC_RMT_SUPPORT_REF_TICK
Support set REF_TICK as the RMT clock source

SOC_RMT_SUPPORT_APB
Support set APB as the RMT clock source

SOC_RMT_CHANNEL_CLK_INDEPENDENT
Can select different source clock for each channel

SOC_RTCIO_PIN_COUNT

SOC_RTCIO_INPUT_OUTPUT_SUPPORTED

SOC_RTCIO_HOLD_SUPPORTED

SOC_RTCIO_WAKE_SUPPORTED

SOC_SDM_GROUPS

SOC_SDM_CHANNELS_PER_GROUP

SOC_SDM_CLK_SUPPORT_APB

SOC_SPI_HD_BOTH_INOUT_SUPPORTED

SOC_SPI_PERIPH_NUM

SOC_SPI_DMA_CHAN_NUM

SOC_SPI_PERIPH_CS_NUM (i)

SOC_SPI_MAX_CS_NUM

SOC_SPI_MAXIMUM_BUFFER_SIZE
SOC_SPI_MAX_PRE_DIVIDER
SOC_SPI_SUPPORT_DDRCLK
SOC_SPI_SLAVE_SUPPORT_SEGMENT_TRANS
SOC_SPI_SUPPORT_CD_SIG
SOC_SPI_SUPPORT_CONTINUOUS_TRANS
SOC_SPI_SUPPORT_CLK_APB
SOC_SPI_SUPPORT_SLAVE_HD_VER2

The SPI Slave half duplex mode has been updated greatly in ESP32-S2.

SOC_SPI_PERIPH_SUPPORT_MULTILINE_MODE (host_id)
SOC_SPI_PERIPH_SUPPORT_CONTROL_DUMMY_OUT

SOC_MEMSPI_IS_INDEPENDENT
SOC_SPI_SUPPORT_OCT
SOC_MEMSPI_SRC_FREQ_80M_SUPPORTED
SOC_MEMSPI_SRC_FREQ_40M_SUPPORTED
SOC_MEMSPI_SRC_FREQ_26M_SUPPORTED
SOC_MEMSPI_SRC_FREQ_20M_SUPPORTED

SOC_SYSTIMER_COUNTER_NUM
SOC_SYSTIMER_ALARM_NUM
SOC_SYSTIMER_BIT_WIDTH_LO
SOC_SYSTIMER_BIT_WIDTH_HI

SOC_TIMER_GROUPS
SOC_TIMER_GROUP_TIMERS_PER_GROUP
SOC_TIMER_GROUP_COUNTER_BIT_WIDTH
SOC_TIMER_GROUP_SUPPORT_XTAL
Chapter 2. API Reference

SOC_TIMER_GROUP_SUPPORT_APB

SOC_TIMER_GROUP_TOTAL_TIMERS

SOC_TOUCH_SENSOR_VERSION
    Hardware version of touch sensor

SOC_TOUCH_SENSOR_NUM
    15 Touch channels

SOC_TOUCH_PROXIMITY_CHANNEL_NUM
    Support touch proximity channel number.

SOC_TOUCH_SAMPLER_NUM
    The sampler number in total, each sampler can be used to sample on one frequency

SOC_TWAI_CONTROLLER_NUM

SOC_TWAI_CLK_SUPPORT_APB

SOC_TWAI_BRP_MIN

SOC_TWAI_BRP_MAX

SOC_TWAI_SUPPORTEDS_RX_STATUS

SOC_UART_NUM

SOC_UART_HP_NUM

SOC_UART_SUPPORT_WAKEUP_INT
    Support UART wakeup interrupt

SOC_UART_SUPPORT_APB_CLK
    Support APB as the clock source

SOC_UART_SUPPORT_REF_TICK
    Support REF_TICK as the clock source

SOC_UART_FIFO_LEN
    The UART hardware FIFO length

SOC_UART_BITRATE_MAX
    Max bit rate supported by UART

SOC_SPIRAM_SUPPORTED

Espressif Systems

Submit Document Feedback

1715

Release master
Chapter 2. API Reference

SOC_SPIRAM_XIP_SUPPORTED
SOC_USB_OTG_PERIPH_NUM
SOC_SHA_DMA_MAX_BUFFER_SIZE
SOC_SHA_SUPPORT_DMA
SOC_SHA_SUPPORT_RESUME
SOC_SHA_CRYPTO_DMA
SOC_SHA_SUPPORT_SHA1
SOC_SHA_SUPPORT_SHA224
SOC_SHA_SUPPORT_SHA256
SOC_SHA_SUPPORT_SHA384
SOC_SHA_SUPPORT_SHA512
SOC_SHA_SUPPORT_SHA512_224
SOC_SHA_SUPPORT_SHA512_256
SOC_SHA_SUPPORT_SHA512_T
SOC_MPI_MEM_BLOCKS_NUM
SOC_MPI_OPERATIONS_NUM
SOC_RSA_MAX_BIT_LEN
SOC_AES_SUPPORT_DMA
SOC_AES_SUPPORT_GCM
SOC_EFUSE_DIS_DOWNLOAD_ICACHE
SOC_EFUSE_DIS_DOWNLOAD_DCACHE
SOC_EFUSE_HARD_DIS_JTAG
SOC_EFUSE_SOFT_DIS_JTAG
Chapter 2. API Reference

SOC_EFUSE_DIS_BOOT_REMAP
SOC_EFUSE_DIS_LEGACY_SPI_BOOT
SOC_EFUSE_DIS_ICACHE
SOC_SECURE_BOOT_V2_RSA
SOC_EFUSE_SECURE_BOOT_KEY_DIGESTS
SOC_EFUSE_REVOKE_BOOT_KEY_DIGESTS
SOC_SUPPORT_SECURE_BOOT_REVOKE_KEY
SOC_FLASH_ENCRYPTED_XTS_AES_BLOCK_MAX
SOC_FLASH_ENCRYPTION_XTS_AES
SOC_FLASH_ENCRYPTION_XTS_AES_OPTIONS
SOC_FLASH_ENCRYPTION_XTS_AES_128
SOC_FLASH_ENCRYPTION_XTS_AES_256
SOC_MEMPROT_CPU_PREFETCH_PAD_SIZE
SOC_MEMPROT_MEM_ALIGN_SIZE
SOC_AES_CRYPTO_DMA
SOC_AES_SUPPORT_AES_128
SOC_AES_SUPPORT_AES_192
SOC_AES_SUPPORT_AES_256
SOC_PHY_DIG_REGS_MEM_SIZE
SOC_WIFI_LIGHT_SLEEP_CLK_WIDTH
SOC_SPI_MEM_SUPPORT_AUTO_WAIT_IDLE
SOC_SPI_MEM_SUPPORT_SW_SUSPEND
SOC_SPI_MEM_SUPPORT_CONFIG_GPIO_BY_EFUSE
SOC_SPI_MEM_SUPPORT_WRAP

SOC_PM_SUPPORT_EXT0_WAKEUP

SOC_PM_SUPPORT_EXT1_WAKEUP

SOC_PM_SUPPORT_EXT_WAKEUP
  Compatible to the old version of IDF

SOC_PM_SUPPORT_WIFI_WAKEUP

SOC_PM_SUPPORT_TOUCH_SENSOR_WAKEUP
  Supports waking up from touch pad trigger

SOC_PM_SUPPORT_WIFI_PD

SOC_PM_SUPPORT_RTC_PERIPH_PD

SOC_PM_SUPPORT_RTC_FAST_MEM_PD

SOC_PM_SUPPORT_RTC_SLOW_MEM_PD

SOC_PM_SUPPORT_RC_FAST_PD

SOC_PM_SUPPORT_VDDSDIO_PD

SOC_CONFIGURABLE_VDDSDIO_SUPPORTED

SOC_CLK_APLL_SUPPORTED

SOC_CLK_RC_FAST_D256_SUPPORTED

SOC_RTC_SLOW_CLK_SUPPORT_RC_FAST_D256

SOC_CLK_RC_FAST_SUPPORT_CALIBRATION

SOC_CLK_XTAL32K_SUPPORTED
  Support to connect an external low frequency crystal

SOC_COEX_HW_PTI

SOC_EXTERNAL_COEX_ADVANCE
  HARDWARE ADVANCED EXTERNAL COEXISTENCE CAPS

SOC_EXTERNAL_COEX_LEADER_TX_LINE
  EXTERNAL COEXISTENCE TX LINE CAPS
Chapter 2. API Reference

**SOC_TEMPERATURE_SENSOR_SUPPORT_FAST_RC**

**SOC_WIFI_HW_TSF**
Support hardware TSF

**SOC_WIFI_FTM_SUPPORT**
Support FTM

**SOC_WIFI_WAPI_SUPPORT**
Support WAPI

**SOC_WIFI_CSI_SUPPORT**
Support CSI

**SOC_WIFI_MESH_SUPPORT**
Support WIFI MESH

**SOC_WIFI_SUPPORT_VARIABLE_BEACON_WINDOW**
Support delta early time for rf phy on/off

**SOC_WIFI_NAN_SUPPORT**
Support WIFI Aware (NAN)

**SOC_ULP_HAS_ADC**

**SOC_PHY_COMBO_MODULE**
Only support Wi-Fi

### 2.9.30 System Time

**Overview**

ESP32-S2 uses two hardware timers for the purpose of keeping system time. System time can be kept by using either one or both of the hardware timers depending on the application’s purpose and accuracy requirements for system time. The two hardware timers are:

- **RTC timer**: This timer allows time keeping in various sleep modes, and can also persist time keeping across any resets (with the exception of power-on resets which reset the RTC timer). The frequency deviation depends on the RTC Timer Clock Sources and affects the accuracy only in sleep modes, in which case the time will be measured at 6.6667 μs resolution.

- **High-resolution timer**: This timer is not available in sleep modes and will not persist over a reset, but has greater accuracy. The timer 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 μs resolution.

The possible combinations of hardware timers used to keep system time are listed below:

- RTC and high-resolution timer (default)
- RTC
- High-resolution timer
- None
Chapter 2. API Reference

It is recommended that users stick to the default option as it provides the highest accuracy. However, users can also select a different setting via the `CONFIG_NEWLIB_TIME_SYSCALL` configuration option.

**RTC Timer Clock Sources**

The RTC timer has the following clock sources:

- **Internal 90 kHz RC oscillator (default)**: Features the lowest Deep-sleep current consumption and no dependence on any external components. However, the frequency stability of this clock source is affected by temperature fluctuations, so time may drift in both Deep-sleep and Light-sleep modes.

- **External 32 kHz crystal**: Requires a 32 kHz crystal to be connected to the XTAL_32K_P and XTAL_32K_N pins. This source provides a better frequency stability at the expense of a slightly higher (by 1 μA) Deep-sleep current consumption.

- **External 32 kHz oscillator at XTAL_32K_P pin**: Allows using 32 kHz clock generated by an external circuit. The external clock signal must be connected to the XTAL_32K_P 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 < Vcm < 0.5xVamp, where Vamp stands for signal amplitude. In this case, the XTAL_32K_P pin cannot be used as a GPIO pin.

- **Internal 8.5 MHz oscillator, divided by 256 (about 33 kHz)**: Provides better frequency stability than the Internal 90 kHz RC oscillator at the expense of a higher (by 5 μA) 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_RTC_CLK_SRC` in project configuration.

More details about the wiring requirements for the external crystal or external oscillator, please refer to [ESP32-S2 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:

```c
gettimeofday
time
timeinfo
asctime
clock
cftime
difftime
gmtime
localtime
mktime
strftime
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 code snippet:

```c
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)
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:

```c
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.

Some lwIP APIs, including SNTP functions, are not thread safe, so it is recommended to use `esp_netif component` when interacting with SNTP module.

To initialize a particular SNTP server and also start the SNTP service, simply create a default SNTP server configuration with a particular server name, then call `esp_netif_sntp_init()` to register that server and start the SNTP service.

```c
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG("pool.ntp.org");
esp_netif_sntp_init(&config);
```

This code automatically performs time synchronization once a reply from the SNTP server is received. Sometimes it is useful to wait until the time gets synchronized, `esp_netif_sntp_sync_wait()` can be used for this purpose:

```c
if (esp_netif_sntp_sync_wait(pdMS_TO_TICKS(10000)) != ESP_OK) {
    printf("Failed to update system time within 10s timeout");
}
```

To configure multiple NTP servers (or use more advanced settings, such as DHCP provided NTP servers), please refer to the detailed description of SNTP API in esp_netif documentation.

The lwIP SNTP library could work in 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()`.

If you want to choose the `SNTP_SYNC_MODE_SMOOTH` mode, please set the `esp_sntp_config::smooth` to `true` in the SNTP configuration struct. Otherwise (and by default) the `SNTP_SYNC_MODE_IMMED` mode will be used.

For setting a callback function that is called when time gets synchronized, use the `esp_sntp_config::sync_cb` field in the configuration struct.

An application with this initialization code periodically synchronizes the time. The time synchronization period is determined by `CONFIG_LWIP_SNTP_UPDATE_DELAY` (the 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 the `protocols/sntp` directory.

Note that it is also possible to use lwIP API directly, but care must be taken to thread safety. Here we list the thread-safe APIs:
- `sntp_set_time_sync_notification_cb()` can be used to set a callback function that notifies of the time synchronization process.
- `sntp_get_sync_status()` and `sntp_set_sync_status()` can be used to get/set time synchronization status.
- `sntp_set_sync_mode()` can be used to set the synchronization mode.
- `esp_sntp_setoperatingmode()` sets the preferred operating mode.
- `esp_sntp_setservername()` configures one SNTP server.

**Timezones**

To set the local timezone, use the following POSIX functions:

1. Call `setenv()` to set the TZ environment variable to the correct value based 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 timezone.

Once these steps are completed, call the standard C library function `localtime()`, and it returns the correct local time taking into account the timezone offset and daylight saving time.

**Year 2036 and 2038 Overflow Issues**

**SNTP/NTP 2036 Overflow** SNTP/NTP timestamps are represented as 64-bit unsigned fixed point numbers, where the first 32 bits represent the integer part, and the last 32 bits represent the fractional part. The 64-bit unsigned fixed point number represents the number of seconds since 00:00 on 1st of January 1900, thus SNTP/NTP times will overflow in the year 2036.

To address this issue, lifetime of the SNTP/NTP timestamps has been extended by convention by using the MSB (bit 0 by convention) of the integer part to indicate time ranges between years 1968 to 2104 (see RFC2030 for more details). This convention is implemented in lwIP library SNTP module. Therefore SNTP-related functions in ESP-IDF are future-proof until year 2104.

**Unix Time 2038 Overflow** Unix time (type `time_t`) was previously represented as a 32-bit signed integer, leading to an overflow in year 2038 (i.e., Y2K38 issue). To address the Y2K38 issue, ESP-IDF uses a 64-bit signed integer to represent `time_t` starting from release v5.0, thus deferring `time_t` overflow for another 292 billion years.

**API Reference**

**Header File**

- `components/lwip/include/apps/esp_sntp.h`
- This header file can be included with:

  ```
  #include "esp_sntp.h"
  ```
- This header file is a part of the API provided by the lwip component. To declare that your component depends on lwip, add the following to your CMakeLists.txt:

  ```
  REQUIREES lwip
  ```
  or

  ```
  PRIV_REQUIREES lwip
  ```
**Chapter 2. API Reference**

**Functions**

```c
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 synchroniza-
tion.

**Parameters** tv – Time received from SNTP server.
```

```c
void sntp_set_sync_mode (sntp_sync_mode_t sync_mode)
Set the sync mode.

Modes allowed: SNTP_SYNC_MODE_IMMED and SNTP_SYNC_MODE_SMOOTH.

**Parameters** sync_mode – Sync mode.
```

```c
sntp_sync_mode_t sntp_get_sync_mode (void)
Get set sync mode.

**Returns**
SNTP_SYNC_MODE_IMMED: Update time immediately.
SNTP_SYNC_MODE_SMOOTH: Smooth time updating.
```

```c
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.

**Returns**
SNTP_SYNC_STATUS_RESET: Reset status. SNTP_SYNC_STATUS_COMPLETED: Time is synchronized. SNTP_SYNC_STATUS_IN_PROGRESS: Smooth time sync in progress.
```

```c
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)
```

```c
void sntp_set_time_sync_notification_cb (sntp_time_sync_cb_t callback)
Set a callback function for time synchronization notification.

**Parameters** callback – a callback function
```

```c
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, oth-
erwise 15 seconds will be set.
```

```c
uint32_t sntp_get_sync_interval (void)
Get the sync interval of SNTP operation.

**Returns** the sync interval
```

```c
bool sntp_restart (void)
Restart SNTP.

**Returns** True - Restart False - SNTP was not initialized yet
```
void **esp_sntp_setoperatingmode**(esp_sntp_operatingmode_t operating_mode)

Sets SNTP operating mode. The mode has to be set before init.

**Parameters**

operating_mode – Desired operating mode

void **esp_sntp_init** (void)

Init and start SNTP service.

void **esp_sntp_stop** (void)

Stops SNTP service.

void **esp_sntp_setserver** (u8_t idx, const ip_addr_t *addr)

Sets SNTP server address.

**Parameters**

- idx – Index of the server
- addr – IP address of the server

void **esp_sntp_setservername** (u8_t idx, const char *server)

Sets SNTP hostname.

**Parameters**

- idx – Index of the server
- server – Name of the server

const char **esp_sntp_getservername** (u8_t idx)

Gets SNTP server name.

**Parameters**

idx – Index of the server

**Returns**

Name of the server

const ip_addr_t **esp_sntp_getserver** (u8_t idx)

Get SNTP server IP.

**Parameters**

idx – Index of the server

**Returns**

IP address of the server

bool **esp_sntp_enabled** (void)

Checks if sntp is enabled.

**Returns**

ture if sntp module is enabled

static inline void **sntp_setoperatingmode** (u8_t operating_mode)

if not build within lwip, provide translating inlines, that will warn about thread safety

static inline void **sntp_servermode_dhcp** (int set_servers_from_dhcp)

static inline void **sntp_setservername** (u8_t idx, const char *server)

static inline void **sntp_init** (void)

static inline const char **sntp_getservername** (u8_t idx)

static inline const ip_addr_t **sntp_getserver** (u8_t idx)

**Macros**

**esp_sntp_sync_time**

Aliases for esp_sntp prefixed API (inherently thread safe)

**esp_sntp_set_sync_mode**
esp_sntp_get_sync_mode

esp_sntp_get_sync_status

esp_sntp_set_sync_status

esp_sntp_set_time_sync_notification_cb

esp_sntp_set_sync_interval

esp_sntp_get_sync_interval

esp_sntp_restart

SNTP_OPMODE_POLL

Type Definitions
typedef void (*sntp_sync_time_cb_t)(struct timeval *tv)
    SNTP callback function for notifying about time sync event.
    
    **Param tv** Time received from SNTP server.

Enumerations
enum sntp_sync_mode_t
    SNTP time update mode.
    
    **Values:**
    
    enumerator **SNTP_SYNC_MODE_IMMED**
    Update system time immediately when receiving a response from the SNTP server.

    enumerator **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:**
    
    enumerator **SNTP_SYNC_STATUS_RESET**

    enumerator **SNTP_SYNC_STATUS_COMPLETED**

    enumerator **SNTP_SYNC_STATUS_IN_PROGRESS**
enum esp_sntp_operatingmode_t
    SNTP operating modes per lwip SNTP module.

    Values:

    enumerator ESP_Sntp_OPMODE_POLL

    enumerator ESP_Sntp_OPMODE_LISTENONLY

2.9.31 Asynchronous Memory Copy

Overview

ESP32-S2 has a DMA engine which can help to offload internal memory copy operations from the CPU in an asynchronous way.

The async memcpy API wraps all DMA configurations and operations. The signature of esp_async_memcpy() is almost the same as the standard libc memcpy function.

The DMA allows multiple memory copy requests to be queued up before the first one is completed, which allows overlap of computation and memory copy. Moreover, it is still possible to know the exact time when a memory copy request is completed by registering an event callback.

Configure and Install Driver

There are several ways to install the async memcpy driver, depending on the underlying DMA engine:

- esp_async_memcpy_install_cpdma() is used to install the async memcpy driver based on the CP DMA engine.
- esp_async_memcpy_install() is a generic API to install the async memcpy driver with a default DMA engine. If the SoC has the CP DMA engine, the default DMA engine is CP DMA. Otherwise, the default DMA engine is AHB DMA.

Driver configuration is described in async_memcpy_config_t:

- backlog: This is used to configure the maximum number of memory copy transactions that can be queued up before the first one is completed. If this field is set to zero, then the default value 4 will be applied.
- sram_trans_align: Declare SRAM alignment for both data address and copy size, set to zero if the data has no restriction in alignment. If set to a quadruple value (i.e., 4X), the driver will enable the burst mode internally, which is helpful for some performance related application.
- psram_trans_align: Declare PSRAM alignment for both data address and copy size. User has to give it a valid value (only 16, 32, 64 are supported) if the destination of memcpy is located in PSRAM. The default alignment (i.e., 16) will be applied if it is set to zero. Internally, the driver configures the size of block used by DMA to access PSRAM, according to the alignment.
- flags: This is used to enable some special driver features.

```c
async_memcpy_config_t config = ASYNC_MEMCOPY_DEFAULT_CONFIG();
// update the maximum data stream supported by underlying DMA engine
config.backlog = 8;
async_memcpy_handle_t driver = NULL;
ESP_ERROR_CHECK(esp_async_memcpy_install(&config, &driver)); // install driver...
```

Send Memory Copy Request

`esp_async_memcpy()` is the API to send memory copy request to DMA engine. It must be called after driver is installed successfully. This API is thread safe, so it can be called from different tasks.

Different from the libc version of `memcpy`, you can optionally pass a callback to `esp_async_memcpy()`, so that you can be notified when the memory copy is finished. Note that the callback is executed in the ISR context, please make sure you will not call any blocking functions in the callback.

The prototype of the callback function is `async_memcpy_isr_cb_t`. The callback function should only return true if it wakes up a high priority task by RTOS APIs like `xSemaphoreGiveFromISR()`.

```c
// Callback implementation, running in ISR context
static bool my_async_memcpy_cb(async_memcpy_handle_t mcp_hdl, async_memcpy_event_t *event, void *cb_args) {
    SemaphoreHandle_t sem = (SemaphoreHandle_t)cb_args;
    BaseType_t high_task_wakeup = pdFALSE;
    xSemaphoreGiveFromISR(semphr, &high_task_wakeup); // high_task_wakeup set to pdTRUE if some high priority task unblocked
    return high_task_wakeup = pdTRUE;
}
```

// Create a semaphore used to report the completion of async memcpy
SemaphoreHandle_t semphr = xSemaphoreCreateBinary();

// Called from user's context
ESP_ERROR_CHECK(esp_async_memcpy(driver_handle, to, from, copy_len, my_async_memcpy_cb, my_semaphore));

// Do something else here
xSemaphoreTake(my_semaphore, portMAX_DELAY); // Wait until the buffer copy is done

Uninstall Driver

`esp_async_memcpy_uninstall()` is used to uninstall asynchronous memcpy driver. It is not necessary to uninstall the driver after each memcpy operation. If you know your application will not use this driver anymore, then this API can recycle the memory and other hardware resources for you.

API Reference

Header File

- components/esp_hw_support/include/esp_async_memcpy.h
- This header file can be included with:

```c
#include "esp_async_memcpy.h"
```

Functions

`esp_err_t esp_async_memcpy_install_cpdma(const async_memcpy_config_t *config, async_memcpy_handle_t *mcp)`

Install async memcpy driver, with CPDMA as the backend.

**Note:** CPDMA is a CPU peripheral, aiming for memory copy.

**Parameters**
- `config` [in] Configuration of async memcpy
- `mcp` [out] Returned driver handle
Returns

- ESP_OK: Install async memcpy driver successfully
- ESP_ERR_INVALID_ARG: Install async memcpy driver failed because of invalid argument
- ESP_ERR_NO_MEM: Install async memcpy driver failed because out of memory
- ESP_FAIL: Install async memcpy driver failed because of other error

```c
esp_err_t esp_async_memcpy_install(const async_memcpy_config_t *config, async_memcpy_handle_t *mcp)
```

Install async memcpy driver with the default DMA backend.

**Note:** On chip with CPDMA support, CPDMA is the default choice. On chip with AHB-GDMA support, AHB-GDMA is the default choice.

Parameters

- config [-in] Configuration of async memcpy
- mcp [-out] Returned driver handle

Returns

- ESP_OK: Install async memcpy driver successfully
- ESP_ERR_INVALID_ARG: Install async memcpy driver failed because of invalid argument
- ESP_ERR_NO_MEM: Install async memcpy driver failed because out of memory
- ESP_FAIL: Install async memcpy driver failed because of other error

```c
esp_err_t esp_async_memcpy_uninstall(async_memcpy_handle_t mcp)
```

Uninstall async memcpy driver.

Parameters mcp [-in] Handle of async memcpy driver that returned from esp_async_memcpy_install

Returns

- ESP_OK: Uninstall async memcpy driver successfully
- ESP_ERR_INVALID_ARG: Uninstall async memcpy driver failed because of invalid argument
- ESP_FAIL: Uninstall async memcpy driver failed because of other error

```c
esp_err_t esp_async_memcpy(async_memcpy_handle_t mcp, void *dst, void *src, size_t n, async_memcpy_isr_cb_t cb_isr, void *cb_args)
```

Send an asynchronous memory copy request.

**Note:** The callback function is invoked in interrupt context, never do blocking jobs in the callback.

Parameters

- mcp [-in] Handle of async memcpy driver that returned from esp_async_memcpy_install
- dst [-in] Destination address (copy to)
- src [-in] Source address (copy from)
- n [-in] Number of bytes to copy
- cb_isr [-in] Callback function, which got invoked in interrupt context. Set to NULL can bypass the callback.
- cb_args [-in] User defined argument to be passed to the callback function

Returns

- ESP_OK: Send memory copy request successfully
- ESP_ERR_INVALID_ARG: Send memory copy request failed because of invalid argument
- ESP_FAIL: Send memory copy request failed because of other error
Structures

struct async_memcpy_event_t
Async memory copy event data.

Public Members

void *data
Event data

struct async_memcpy_config_t
Type of async memcpy configuration.

Public Members

uint32_t backlog
Maximum number of transactions that can be prepared in the background

size_t sram_trans_align
DMA transfer alignment (both in size and address) for SRAM memory

size_t psram_trans_align
DMA transfer alignment (both in size and address) for PSRAM memory

uint32_t flags
Extra flags to control async memcpy feature

Macros

ASYNC_MEMCPY_DEFAULT_CONFIG()
Default configuration for async memcpy.

Type Definitions

typedef struct async_memcpy_context_t *async_memcpy_handle_t
Async memory copy driver handle.

typedef bool (*async_memcpy_isr_cb_t)(async_memcpy_handle_t mcp_hdl, async_memcpy_event_t *event, void *cb_args)
Type of async memcpy interrupt callback function.

Note: User can call OS primitives (semaphore, mutex, etc) in the callback function. Keep in mind, if any OS primitive wakes high priority task up, the callback should return true.

Param mcp_hdl Handle of async memcpy
Param event Event object, which contains related data, reserved for future
Param cb_args User defined arguments, passed from esp_async_memcpy function
Return Whether a high priority task is woken up by the callback function
2.9.32 ULP Coprocessor Programming

The Ultra Low Power (ULP) coprocessor is a simple finite state machine (FSM) 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 the 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. This coprocessor is referred to as ULP FSM in ESP-IDF.

ESP32-S2 provides a second type of ULP coprocessor which is based on a RISC-V instruction set architecture. For details regarding ULP RISC-V refer ULP-RISC-V Coprocessor.

Installing the Toolchain

The ULP FSM 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 FSM toolchain will already be installed.

Programming ULP FSM

The ULP FSM can be programmed using the supported instruction set. Alternatively, the ULP FSM coprocessor can also be programmed using C Macros on the main CPU. These two methods are described in the following section:

ESP32-S2 ULP Coprocessor Instruction Set

This document provides details about the instructions used by ESP32-S2 ULP FSM coprocessor assembler.

ULP FSM 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 8 K 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 FSM coprocessor’s JUMP, ST, LD family of instructions expect the address argument to be expressed in the following way depending on the type of address argument used:

- When the address argument is presented as a label then the instruction expects the address to be expressed as 32-bit words.

Consider the following example program:

```c
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 `R1` to be expressed in 32-bit words. To account for this common use case, the assembler will convert the address of label `loop` from bytes to words when generating the `MOVE` instruction. Hence, the code generated code will be equivalent to:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>000c</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>MOVE R1, 4</td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td>JUMP R1</td>
<td></td>
</tr>
</tbody>
</table>

- 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:

```assembly
.set val, 0x10
MOVE R1, val
```

In this case, value loaded into `R1` will be `0x10`.

However, when an immediate value is used as an offset in `LD` and `ST` instructions, the assembler considers the address argument in bytes and converts it to 32-bit words before executing the instruction:

```
ST R1, R2, 4 // offset = 4 bytes; Mem[R2 + 4 / 4] = R1
```

In this case, the value in `R1` is stored at the memory location pointed by `[R2 + offset / 4]`.

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.

The detailed description of all instructions is presented below:

**Difference Between ESP32 ULP and ESP32-S2 ULP Instruction Sets**  Compared to the ESP32 ULP FSM coprocessor, the ESP32-S2 ULP FSM coprocessor has an extended instruction set. The ESP32-S2 ULP FSM is not binary compatible with ESP32 ULP FSM, but a ESP32 ULP FSM assembled program is expected to work on the ESP32-S2 ULP FSM after rebuilding. The list of the new instructions that was added to the ESP32-S2 ULP FSM is: LDL, LDH, STL, STH, ST32, STO, STI, STI32.

**NOP - No Operation**  Syntax

NOP

Operands
None

Cycles
2 cycle to execute, 4 cycle to fetch next instruction

Description
No operation is performed. Only the PC is incremented.

Example:

```
1:  NOP
```

**ADD - Add to Register**  Syntax

ADD Rdst, Rsoc1, Rsoc2

ADD Rdst, Rsoc1, imm

Operands
- **Rdst** - Register R[0..3]
- **Rsoc1** - Register R[0..3]
- **Rsoc2** - 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 the result in 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)
SUB - Subtract from Register

Syntax

SUB Rdst, Rsrcl, Rsrcl2

SUB Rdst, Rsrcl, imm

Operands

• Rdst - Register R[0..3]
• Rsrcl - Register R[0..3]
• Rsrcl2 - 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 a 16-bit signed value from a source register, and stores the 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 - Bitwise Logical AND of Two Operands

Syntax

AND Rdst, Rsrcl, Rsrcl2

AND Rdst, Rsrcl, imm

Operands

• Rdst - Register R[0..3]
• Rsrcl - Register R[0..3]
• Rsrcl2 - Register R[0..3]
• Imm - 16-bit signed value

Cycles

2 cycles to execute, 4 cycles to fetch next instruction

Description

The instruction does a bitwise logical AND of a source register and another source register or a 16-bit signed value and stores the result to the destination register.

Examples:
Chapter 2. API Reference

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 - Bitwise Logical OR of Two Operands**

**Syntax**

OR Rdst, Rsrcl, Rsrc2

OR Rdst, Rsrcl, imm

**Operands**

- **Rdst** - Register R[0..3]
- **Rsrcl** - 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 a bitwise logical OR of a source register and another source register or a 16-bit signed value and stores the 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, Rsrcl, Rsrcl2

LSH Rdst, Rsrcl, imm

**Operands**

- **Rdst** - Register R[0..3]
- **Rsrcl** - 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 a logical shift to left of the source register by the number of bits from another source register or a 16-bit signed value and stores the result to the destination register.

**Note:** Shift operations which are greater than 15 bits have undefined results.

### 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, Rscl, Rscl2`

`RSH Rdst, Rscl, 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 a logical shift to right of a source register by the number of bits from another source register or a 16-bit signed value and stores the result to the destination register.

**Note:** Shift operations which are greater than 15 bits have undefined results.

### 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, Rscl`

`MOVE Rdst, imm`

**Operands**
Chapter 2. API Reference

- **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 moves the value from the source register or a 16-bit signed value to the destination register.

**Note:** 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. See the section *Note About Addressing* for more details.

**Examples**:

<table>
<thead>
<tr>
<th>No</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MOVE R1, R2</td>
<td>R1 = R2</td>
</tr>
<tr>
<td>2</td>
<td>MOVE R1, 0x03</td>
<td>R1 = 0x03</td>
</tr>
<tr>
<td>3</td>
<td>.set value1, 0x03</td>
<td>constant value1=0x03</td>
</tr>
<tr>
<td></td>
<td>MOVE R1, value1</td>
<td>R1 = value1</td>
</tr>
<tr>
<td>4</td>
<td>.global label</td>
<td>declaration of label</td>
</tr>
<tr>
<td></td>
<td>MOVE R1, label</td>
<td>R1 = address_of(label) / 4</td>
</tr>
<tr>
<td></td>
<td>label: nop</td>
<td>definition of label</td>
</tr>
</tbody>
</table>

**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** – 13-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) OR’d with Rdst (0..3):

\[
\text{Mem[Rdst + offset / 4][31:0]} = \{\text{PC[10:0]}, 3'b0, \text{Rdst, Rsrc[15:0]}}
\]

The application can use the higher 16 bits to determine which instruction in the ULP program has written any particular word into memory.

**Note:** Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section *Note About Addressing* for more details.

**Examples:**
STL – Store Data to the Lower 16 Bits of 32-bit Memory  

**Syntax**

\[ \text{STL} \text{ Rs src, Rd dst, offset, Label} \]

**Operands**

- **Rs src** – Register R[0..3], holds the 16-bit value to store
- **Rd dst** – 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 Rs src to the lower half-word of the memory with address \([\text{Rd dst + offset / 4}]\):

\[
\text{Mem[Rs src]}{15:0} = \text{Mem[Rd dst + offset / 4]{15:0}}
\]

The ST and the STL commands can be used interchangeably and have been provided to maintain back-ward compatibility with previous versions of the ULP core.

**Note:** Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section Note About Addressing for more details.

<table>
<thead>
<tr>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: STL R1, R2, 0x12 // MEM[R2 + 0x12 / 4] = R1</td>
</tr>
</tbody>
</table>
| 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 / 4] = R1  
|             // MEM[Addr1 + 0] will be 32'h600001 |
| 3: MOVE R1, 1     // R1 = 1  
| STL R1, R2, 0x12, 1 // MEM[R2 + 0x12 / 4] = 0xxxxx4001 |

STH – Store data to the higher 16 bits of 32-bit memory  

**Syntax**

\[ \text{STH} \text{ Rs src, Rd dst, offset, Label} \]

**Operands**

<table>
<thead>
<tr>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: STL R1, R2, 0x12 // MEM[R2 + 0x12 / 4] = R1</td>
</tr>
</tbody>
</table>
| 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 / 4] = R1  
|             // MEM[Addr1 + 0] will be 32'h600001 |
| 3: MOVE R1, 1     // R1 = 1  
| STL R1, R2, 0x12, 1 // MEM[R2 + 0x12 / 4] = 0xxxxx4001 |
Chapter 2. API Reference

• **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 upper half-word of memory with address [Rdst + offset / 4]:

```
Mem[Rdst + offset / 4]{31:16} = {Rsrc[15:0]}
Mem[Rdst + offset / 4]{31:16} = {Label[1:0], Rsrc[13:0]}
```

**Note:** Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section **Note About Addressing** for more details.

**Examples:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STH R1, R2, 0x12 // MEM[R2 + 0x12 / 4]{31:16} = R1</td>
</tr>
</tbody>
</table>
| 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 / 4] = R1  
|  | MEM[Addr1 + 0] will be 32'h0001xxxx  
| 3 | MOVE R1, 1 // R1 = 1  
|  | STH R1, R2, 0x12, 1 // MEM[R2 + 0x12 / 4] 0x4001xxxx |

**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-bit memory with address [Rdst + offset / 4]:

```
Mem[Rdst + offset / 4]{31:0} = (PC[10:0], 0[2:0], Label[1:0], Rsrc[15:0])
```

**Note:** Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section **Note About Addressing** for more details.
Examples:

1:    ST32  R1, R2, 0x12, 0 // MEM[R2 + 0x12 / 4][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
            ST32 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

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 sets the 16-bit value to the offset register:

\[ offset = value / 4 \]

Note: Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section Note About Addressing for more details.

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 Rsrcto the lower and upper half-word of memory with address [Rdst + offset / 4].

The offset value is auto incremented when the STI instruction is called twice. Make sure to execute the STO instruction to set the offset value before executing the STI instruction:

\[
\text{Mem}[\text{Rdst} + \text{offset} \div 4]\{15:0/31:16\} = \{\text{Rsrc}[15:0]\} \\
\text{Mem}[\text{Rdst} + \text{offset} \div 4]\{15:0/31:16\} = \{\text{Label}[1:0],\text{Rsrc}[13:0]\}
\]

Examples:

\begin{verbatim}
1:   STO   4      // Set offset to 4 
    STI R1, R2  // MEM[R2 + 4 / 4][15:0] = R1 
    STI R1, R2  // MEM[R2 + 4 / 4][31:16] = R1 
      \[~1 \text{ word}\] 
    STI R1, R2  // MEM[R2 + 8 / 4][15:0] = R1 
    STI R1, R2  // MEM[R2 + 8 / 4][31:16] = R1 
\end{verbatim}

STI32 – Store 32-bits data to the 32-bits memory with auto increment of adress offset  

\textbf{Syntax}

STI32 Rsrc, Rdst, Label

\textbf{Operands}

- \texttt{Rsrc} – Register R[0..3], holds the 16-bit value to store
- \texttt{Rdst} – Register R[0..3], address of the destination, in 32-bit words
- \texttt{Label} – 2-bit user defined unsigned value

\textbf{Cycles}

4 cycles to execute, 4 cycles to fetch next instruction

\textbf{Description}

The instruction stores 11 bits of the PC value, label value and the 16-bit value of Rsrc to the 32-bit memory with address [Rdst + offset / 4].

The offset value is auto incremented each time the STI32 instruction is called. Make sure to execute the STO instruction to set the offset value before executing the STI32 instruction:

\[
\text{Mem}[\text{Rdst} + \text{offset} \div 4]\{31:0\} = \{\text{PC}[10:0],0[2:0],\text{Label}[1:0],\text{Rsrc}[15:0]\}
\]

Examples:

\begin{verbatim}
1:   STO 0x12 
    STI32 R1, R2, 0 // MEM[R2 + 0x12 / 4][31:0] = \{PC[10:0],0[2:0],Label[1:0],Rsrc[15:0]\} 
      \[~\text{Label}[1:0],\text{Rsrc}[15:0]\} 
      \[~\text{word}\] 
    STI32 R1, R2, 0 // MEM[R2 + 0x16 / 4][31:0] = \{PC[10:0],0[2:0],Label[1:0],Rsrc[15:0]\}
\end{verbatim}

LD – Load Data from the Memory  

\textbf{Syntax}

LD Rdst, Rsrc, offset

\textbf{Operands}

- \texttt{Rdst} – Register R[0..3], destination
- \texttt{Rsrc} – Register R[0..3], holds address of destination, in 32-bit words
- \texttt{Offset} – 13-bit signed value, offset in bytes

\textbf{Cycles}
4 cycles to execute, 4 cycles to fetch next instruction

**Description**

The instruction loads the lower 16-bit half-word from memory with address \([Rsrc + \text{offset} / 4]\) into the destination register \(Rdst\):

\[
Rdst[15:0] = \text{Mem}[Rsrc + \text{offset} / 4][15:0]
\]

**Note:** Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section *Note About Addressing* for more details.

**Examples:**

1: LDL R1, R2, 0x12   // R1 = MEM[R2 + 0x12 / 4]

2: .data            // Data section definition
    Addr1: .word 123   // Define label Addr1 16 bit
    .set offs, 0x00   // Define constant offs
    .text
    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 + offs]

LDL –Load Data from the Lower Half-Word of the 32-bit Memory  Syntax

**LDL Rdst, Rsrc, offset**

**Operands**

- \(Rdst\) – Register \(R[0..3]\), destination
- \(Rsrc\) – Register \(R[0..3]\), holds address of destination, in 32-bit words
- \(\text{Offset}\) – 13-bit signed value, offset in bytes

**Cycles**

4 cycles to execute, 4 cycles to fetch next instruction

**Description**

The instruction loads the lower 16-bit half-word from memory with address \([Rsrc + \text{offset} / 4]\) into the destination register \(Rdst\):

\[
Rdst[15:0] = \text{Mem}[Rsrc + \text{offset} / 4][15:0]
\]

The LD and the LDL commands can be used interchangeably and have been provided to maintain backward compatibility with previous versions of the ULP core.

**Note:** Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section *Note About Addressing* for more details.

**Examples:**

1: LDL R1, R2, 0x12   // R1 = MEM[R2 + 0x12 / 4]

2: .data            // Data section definition
    Addr1: .word 123   // Define label Addr1 16 bit
    .set offs, 0x00   // Define constant offs

(continues on next page)
## LDH – Load data from upper half-word of the 32-bit memory

### Syntax

**LDH Rdst, Rsfc, offset**

### Operands

- **Rdst** – Register R[0..3], destination
- **Rsfc** – Register R[0..3], holds address of destination, in 32-bit words
- **Offset** – 13-bit signed value, offset in bytes

### Cycles

4 cycles to execute, 4 cycles to fetch next instruction

### Description

The instruction loads the upper 16-bit half-word from memory with address [Rsfc + offset / 4] into the destination register Rdst:

\[ \text{Rdst}[15:0] = \text{Mem}[\text{Rsfc} + \text{offset} / 4][15:0] \]

### Note:

Note that the offset specified in bytes is converted to a 32-bit word offset before execution. See the section **Note About Addressing** for more details.

### Examples:

1:  

```
LDH R1, R2, 0x12  // R1 = MEM[R2 + 0x12 / 4]
```

2:  

```
.data
Addr1: .word 0x12345678  // Define label Addr1 16 bit
.set offs, 0x00  // Define constant offs
.text
MOVE R1, 1  // R1 = 1
MOVE R2, Addr1  // R2 = Addr1 / 4 (address of label is...
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:

```plaintext
// 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 depends 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
   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**
2 cycles to execute, 2 cycles to fetch next instruction

**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:**
Chapter 2. API Reference

### STAGE_RST

Reset stage count register

### STAGE_INC

#### Syntax

STAGE_INC Value

#### Operands

- **Value**: 8 bits value

#### Cycles

2 cycles to execute, 4 cycles to fetch next instruction

#### Description

The instruction increments the stage count register by the 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

#### Syntax

STAGE_DEC Value

#### Operands

- **Value**: 8 bits value

#### Cycles

2 cycles to execute, 4 cycles to fetch next instruction

#### Description

The instruction decrements the stage count register by the 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
Chapter 2. API Reference

Description

The instruction halts the ULP coprocessor and restarts the ULP wakeup timer, if it is enabled.

Examples:

```plaintext
1: HALT       // Halt the coprocessor
```

**WAKE – Wake up the Chip**  

**Syntax**

```plaintext
WAKE
```

**Operands**

No operands

**Cycles**

2 cycles to execute, 4 cycles to fetch next instruction

**Description**

The instruction sends an interrupt from the ULP coprocessor to the 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:** Note that before using WAKE instruction, ULP program may need 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). If the WAKE instruction is intended to be used while the main CPU is not in sleep mode then the RTC_CNTL_MAIN_STATE_IN_IDLE (bit 27) of RTC_CNTL_LOW_POWER_ST_REG can be used to check whether main CPU is in normal mode or sleep mode.

**Examples:**

```plaintext
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.
```

```plaintext
1: check_wakeup:            // Read RTC_CNTL_RDY_FOR_WAKEUP and RTC_CNTL_MAIN_STATE_IN_IDLE bit
    READ_RTC_REG(RTC_CNTL_LOW_POWER_ST_REG, 27, 1)
    MOVE r1, r0                // Copy result in to r1
    READ_RTC_FIELD(RTC_CNTL_LOW_POWER_ST_REG, RTC_CNTL_RDY_FOR_WAKEUP)
    OR r0, r0, r1
    JUMP check_wakeup, eq      // Retry until either of the bit are set
WAKE                  // Trigger wake up
HALT                   // Stop the ULP program
```

**WAIT – Wait Some Number of Cycles**  

**Syntax**

```plaintext
WAIT Cycles
```

Espressif Systems  

1746  

Release master  

Submit Document Feedback
Chapter 2. API Reference

Operands

- **Cycles** – number of cycles for wait

**Cycles**

\[ 2 + \text{Cycles} \] 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**

\[ 2 + \text{Wait_Delay} + 3 \times \text{TSENS_CLK} \] 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. If the user passes Mux value 1, then ADC pad 0 gets used.

**Cycles**

\[ 23 + \max(1, \text{SAR\_AMP\_WAIT1}) + \max(1, \text{SAR\_AMP\_WAIT2}) + \max(1, \text{SAR\_AMP\_WAIT3}) + \text{SAR\_SAMPLE\_CYCLE} + \text{SAR\_SAMPLE\_BIT} \] cycles to execute, 4 cycles to fetch next instruction

**Description**

The instruction makes measurements from ADC.

**Examples:**

```
.. only:: esp32
```
1: ADC R1, 0, 1 // Measure value using ADC1 channel 0 and store result into R1
1: ADC R1, 0, 1 // Measure value using ADC1 pad 2 and store result into R1

**REG_RD – Read from Peripheral Register**

**Syntax**

\[
\text{REG_RD Addr, High, Low}
\]

**Operands**

- \( \text{Addr} \) – Register address, in 32-bit words
- \( \text{High} \) – Register end bit number
- \( \text{Low} \) – 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:

\[
R0 = \text{REG}[\text{Addr}][\text{High}:\text{Low}].
\]

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the register, as seen from the ULP, can be calculated from the address of the same register on the PeriBUS1 as follows:

\[
\text{addr}_\text{ulp} = (\text{addr}_\text{peribus1} - \text{DR_REG_RTCCNTL_BASE}) / 4
\]

**Examples:**

1: REG_RD 0x120, 7, 4 // load 4 bits: R0 = (12'b0, REG[0x120][7:4])

**REG_WR – Write to Peripheral Register**

**Syntax**

\[
\text{REG_WR Addr, High, Low, Data}
\]

**Operands**

- \( \text{Addr} \) – Register address, in 32-bit words.
- \( \text{High} \) – Register end bit number
- \( \text{Low} \) – Register start bit number
- \( \text{Data} \) – 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:

\[
\text{REG}[\text{Addr}][\text{High}:\text{Low}] = \text{data}.
\]

This instruction can access registers in RTC_CNTL, RTC_IO, SENS, and RTC_I2C peripherals. Address of the register, as seen from the ULP, can be calculated from the address of the same register on the PeriBUS1 as follows:

\[
\text{addr}_\text{ulp} = (\text{addr}_\text{peribus1} - \text{DR_REG_RTCCNTL_BASE}) / 4
\]

**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:

```c
#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:

```c
#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:

```c
#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:

```c
#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)
```

Programming ULP FSM Coprocessor Using C Macros (Legacy)

In addition to the existing binutils port for the ESP32-S2 ULP coprocessor, it is possible to generate programs for the ULP FSM coprocessor by embedding assembly-like macros into an ESP32-S2 application. Here is an example how this can be done:

```c
const ulp_insn_t program[] = {
    I_MVRI(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, R2, 2), // R2 -> RTC_SLOW_MEM[R2 + 2]
    I_HALT()
};
```

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, R1, ...).
—R3) and literal constants. See the API reference section at the end of this guide 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.

**Note:** Load, store and move instructions use **addresses expressed in 32-bit words**. Address 0 corresponds to the first word of RTC_SLOW_MEM. This is different to how address arguments are handled in assembly code of the same instructions. See the section **Note About Addressing** for more details for reference.

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 extra ulp_insn_t token which contains the label number.

Here is an example of using labels and branches:

```c
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, ssize);
ulp_run(load_addr);
```

**API Reference**

**Header File**

- components/ulp/ulp_fsm/include/esp32s2/ulp.h
- This header file can be included with:

  ```c
  #include "ulp.h"
  ```
  
  This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakeLists.txt:

  ```cmake
  REQUIRES ulp
  ```

  or
Functions
static inline uint32_t SOC_REG_TO_ULP_PERIPH_SEL (uint32_t reg)
    Map SoC peripheral register to periph_sel field of RD_REG and WR_REG instructions.

Parameters reg peripheral register in RTC_CNTL_, RTC_IO_, SENS_, RTC_I2C peripher-
als.

Returns periph_sel value for the peripheral to which this register belongs.

Unions
union ulp_insn
    #include <ulp.h> Instruction format structure.

All ULP instructions are 32 bit long. This union contains field layouts used by all of the supported instructions.
This union also includes a special “macro” instruction layout. This is not a real instruction which can be executed
by the CPU. It acts as a token which is removed from the program by the ulp_process_macros_and_load
function.

These structures are not intended to be used directly. Preprocessor definitions provided below fill the fields of
these structure with the right arguments.

Public Members

uint32_t cycles
    Number of cycles to sleep
    TBD, cycles used for measurement

uint32_t unused
    Unused

uint32_t opcode
    Opcode (OPCODE_DELAY)
    Opcode (OPCODE_ST)
    Opcode (OPCODE_LD)
    Opcode (OPCODE_HALT)
    Opcode (OPCODE_BRANCH)
    Opcode (OPCODE_ALU)
    Opcode (OPCODE_WR_REG)
    Opcode (OPCODE_RD_REG)
    Opcode (OPCODE_ADC)
    Opcode (OPCODE_TSENS)
    Opcode (OPCODE_I2C)
    Opcode (OPCODE_END)
    Opcode (OPCODE_MACRO)
struct *ulp_insn::[anonymous] delay

Format of DELAY instruction

uint32_t dreg
  Register which contains data to store
  Register where the data should be loaded to
  Register which contains target PC, expressed in words (used if .reg == 1)
  Destination register
  Register where to store ADC result
  Register where to store temperature measurement result

uint32_t sreg
  Register which contains address in RTC memory (expressed in words)
  Register with operand A

uint32_t label
  Data label, 2-bit user defined unsigned value
  Label number

uint32_t upper
  0: write the low half-word; 1: write the high half-word

uint32_t wr_way
  0: write the full-word; 1: with the label; 3: without the label

uint32_t unused1
  Unused

uint32_t offset
  Offset to add to sreg
  Absolute value of target PC offset w.r.t. current PC, expressed in words

uint32_t unused2
  Unused

uint32_t sub_opcode
  Sub opcode (SUB_OPCODE_ST)
  Sub opcode (SUB_OPCODE_BX)
  Sub opcode (SUB_OPCODE_B)
  Sub opcode (SUB_OPCODE_ALU_REG)
  Sub opcode (SUB_OPCODE_ALU_IMM)
  Sub opcode (SUB_OPCODE_ALU_CNT)
  Sub opcode (SUB_OPCODE_WAKEUP)
  SUB_OPCODE_MACRO_LABEL or SUB_OPCODE_MACRO_BRANCH
struct `ulp_insn`::[anonymous] `st`
Format of ST instruction

uint32_t `rd_upper`
0: read the high half-word; 1: read the low half-word

struct `ulp_insn`::[anonymous] `ld`
Format of LD instruction

struct `ulp_insn`::[anonymous] `halt`
Format of HALT instruction

uint32_t `addr`
Target PC, expressed in words (used if .reg == 0)
Address within either RTC_CNTL, RTC_IO, or SARADC

uint32_t `reg`
Target PC in register (1) or immediate (0)

uint32_t `type`
Jump condition (BX_JUMP_TYPE_xxx)

struct `ulp_insn`::[anonymous] `bx`
Format of BRANCH instruction (absolute address)

uint32_t `imm`
Immediate value to compare against
Immediate value of operand B
Immediate value

uint32_t `cmp`
Comparison to perform: B_CMP_L or B_CMP_GE

uint32_t `sign`
Sign of target PC offset: 0: positive, 1: negative

struct `ulp_insn`::[anonymous] `b`
Format of BRANCH instruction (relative address)

uint32_t `treg`
Register with operand B

uint32_t `sel`
Operation to perform, one of ALU_SEL_xxx

struct `ulp_insn`::[anonymous] `alu_reg`
Format of ALU instruction (both sources are registers)
struct ulp_insn::[anonymous] alu_imm
  Format of ALU instruction (one source is an immediate)

uint32_t unused3
  Unused

struct ulp_insn::[anonymous] alu_cnt
  Format of ALU instruction with stage count register and an immediate

uint32_t periph_sel
  Select peripheral: RTC_CNTL (0), RTC_IO (1), SARADC (2)

uint32_t data
  8 bits of data to write
  Data to read or write

uint32_t low
  Low bit

uint32_t high
  High bit

struct ulp_insn::[anonymous] wr_reg
  Format of WR_REG instruction

struct ulp_insn::[anonymous] rd_reg
  Format of RD_REG instruction

uint32_t mux
  Select SARADC pad (mux + 1)

uint32_t sar_sel
  Select SARADC0 (0) or SARADC1 (1)

struct ulp_insn::[anonymous] adc
  Format of ADC instruction

uint32_t wait_delay
  Cycles to wait after measurement is done

uint32_t reserved
  Reserved, set to 0

struct ulp_insn::[anonymous] tsens
  Format of TSENS instruction

uint32_t i2c_addr
  I2C slave address
Chapter 2. API Reference

uint32_t low_bits
    TBD

uint32_t high_bits
    TBD

uint32_t i2c_sel
    TBD, select reg_i2c_slave_address[7:0]

uint32_t rw
    Write (1) or read (0)

struct ulp_insn::[anonymous] i2c
    Format of I2C instruction

uint32_t wakeup
    Set to 1 to wake up chip

struct ulp_insn::[anonymous] end
    Format of END instruction with wakeup

struct ulp_insn::[anonymous] macro
    Format of tokens used by LABEL and BRANCH macros

Macros

R0
    general purpose register 0

R1
    general purpose register 1

R2
    general purpose register 2

R3
    general purpose register 3

OPCODE_WR_REG
    Instruction: write peripheral register (RTC_CNTL/RTC_IO/SARADC) (not implemented yet)

OPCODE_RD_REG
    Instruction: read peripheral register (RTC_CNTL/RTC_IO/SARADC) (not implemented yet)

RD_REG_PERIPH_RTC_CNTL
    Identifier of RTC_CNTL peripheral for RD_REG and WR_REG instructions

RD_REG_PERIPH_RTC_IO
    Identifier of RTC_IO peripheral for RD_REG and WR_REG instructions
RD_REG_PERIPH_SENS
Identifier of SARADC peripheral for RD_REG and WR_REG instructions

RD_REG_PERIPH_RTC_I2C
Identifier of RTC_I2C peripheral for RD_REG and WR_REG instructions

OPCODE_I2C
Instruction: read/write I2C (not implemented yet)

OPCODE_DELAY
Instruction: delay (nop) for a given number of cycles

OPCODE_ADC
Instruction: SAR ADC measurement (not implemented yet)

OPCODE_ST
Instruction: store indirect to RTC memory

SUB_OPCODE_ST_AUTO
Automatic Storage Mode - Access continuous addresses. Use SUB_OPCODE_ST_OFFSET to configure the initial address before using this instruction.

SUB_OPCODE_ST_OFFSET
Automatic Storage Mode - Configure the initial address.

SUB_OPCODE_ST
Manual Storage Mode. Store 32 bits, 16 MSBs contain PC, 16 LSBs contain value from source register

OPCODE_ALU
Arithmetic instructions

SUB_OPCODE_ALU_REG
Arithmetic instruction, both source values are in register

SUB_OPCODE_ALU_IMM
Arithmetic instruction, one source value is an immediate

SUB_OPCODE_ALU_CNT
Arithmetic instruction between counter register and an immediate (not implemented yet)

ALU_SEL_ADD
Addition

ALU_SEL_SUB
Subtraction

ALU_SEL_AND
Logical AND
Chapter 2. API Reference

**ALU_SEL_OR**
Logical OR

**ALU_SEL_MOV**
Copy value (immediate to destination register or source register to destination register)

**ALU_SEL_LSH**
Shift left by given number of bits

**ALU_SEL_RSH**
Shift right by given number of bits

**ALU_SEL_STAGE_INC**
Increment stage count register

**ALU_SEL_STAGE_DEC**
Decrement stage count register

**ALU_SEL_STAGE_RST**
Reset stage count register

**OPCODE_BRANCH**
Branch instructions

**SUB_OPCODE_B**
Branch to a relative offset

**SUB_OPCODE_BX**
Branch to absolute PC (immediate or in register)

**SUB_OPCODE_BS**
Branch to a relative offset by comparing the stage_cnt register

**BX_JUMP_TYPE_DIRECT**
Unconditional jump

**BX_JUMP_TYPE_ZERO**
Branch if last ALU result is zero

**BX_JUMP_TYPE_OVF**
Branch if last ALU operation caused and overflow

**B_CMP_L**
Branch if R0 is less than an immediate

**B_CMP_G**
Branch if R0 is greater than an immediate
B_CMP_E
Branch if R0 is equal to an immediate

BS_CMP_L
Branch if stage_cnt is less than an immediate

BS_CMP_GE
Branch if stage_cnt is greater than or equal to an immediate

BS_CMP_LE
Branch if stage_cnt is less than or equal to an immediate

OPCODE_END
Stop executing the program

SUB_OPCODE_END
Stop executing the program and optionally wake up the chip

SUB_OPCODE_SLEEP
Stop executing the program and run it again after selected interval

OPCODE_TSENS
Instruction: temperature sensor measurement (not implemented yet)

OPCODE_HALT
Halt the coprocessor

OPCODE_LD
Indirect load lower 16 bits from RTC memory

OPCODE_MACRO
Not a real opcode. Used to identify labels and branches in the program

SUB_OPCODE_MACRO_LABEL
Label macro

SUB_OPCODE_MACRO_BRANCH
Branch macro

SUB_OPCODE_MACRO_LABELPC
Label pointer macro

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_WR_REG** *(reg, low_bit, high_bit, val)*  
Write literal value to a peripheral register  
\[ \text{reg}[\text{high_bit} : \text{low_bit}] = \text{val} \]  
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  
\[ R0 = \text{reg}[\text{high_bit} : \text{low_bit}] \]  
This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

**I_WR_REG_BIT** *(reg, shift, val)*  
Set or clear a bit in the peripheral register.  
Sets bit \((1 \ll \text{shift})\) of register reg to value val. This instruction can access RTC_CNTL_, RTC_IO_, SENS_, and RTC_I2C peripheral registers.

**I_WAKE** ()  
Wake the SoC from deep sleep.  
This instruction initiates wake up from deep sleep. Use esp_deep_sleep_enable_ulp_wakeup to enable deep sleep wakeup triggered by the ULP before going into deep sleep. Note that ULP program will still keep running until the I_HALT instruction, and it will still be restarted by timer at regular intervals, even when the SoC is woken up.  
To stop the ULP program, use I_HALT 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_TSENS** *(reg_dest, delay)*  
Perform temperature sensor measurement and store it into reg_dest.  
Delay can be set between 1 and \((1 \ll 14) - 1\). Higher values give higher measurement resolution.

**I_ADC** *(reg_dest, adc_idx, pad_idx)*  
Perform ADC measurement and store result in reg_dest.  
adc_idx selects ADC (0 or 1). pad_idx selects ADC pad (0 - 7).

**I_ST_MANUAL** *(reg_val, reg_addr, offset_, label_, upper_, wr_way_)*  
Store lower half-word, upper half-word or full-word data from register reg_val into RTC memory address.  
This instruction can be used to write data to discontinuous addresses in the RTC_SLOW_MEM. The value is written to an offset calculated by adding the value of reg_addr register and offset_field (this offset is expressed in 32-bit words). The storage method is dictated by the wr_way and upper_field settings as summarized in the following table:

<table>
<thead>
<tr>
<th>*</th>
<th>------------</th>
<th>-------</th>
<th>------------------------------------------</th>
<th>------------------------------------------</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>wr_way</td>
<td>upper</td>
<td></td>
<td></td>
<td>operation</td>
</tr>
</tbody>
</table>

(continues on next page)
### SUB_OPCODE_ST

```
SUB_OPCODE_ST = manual_en:1, offset_set:0, wr_auto:0
```

**I_ST** (*reg_val, reg_addr, offset_*)

Store value from register `reg_val` into RTC memory.

I_ST() instruction provides backward compatibility for code written for esp32 to be run on esp32s2. This instruction is equivalent to calling I_ST_MANUAL() instruction with `label = 0`, `upper = 0` and `wr_way = 3`.

**I_STL** (*reg_val, reg_addr, offset_*)

Store value from register `reg_val` to lower 16 bits of the RTC memory address.

This instruction is equivalent to calling I_ST_MANUAL() instruction with `label = 0`, `upper = 0` and `wr_way = 3`.

**I_STH** (*reg_val, reg_addr, offset_*)

Store value from register `reg_val` to upper 16 bits of the RTC memory address.

This instruction is equivalent to calling I_ST_MANUAL() instruction with `label = 0`, `upper = 1` and `wr_way = 3`.

**I_ST32** (*reg_val, reg_addr, offset_, label_*)

Store value from register `reg_val` to full 32 bit word of the RTC memory address.

This instruction is equivalent to calling I_ST_MANUAL() instruction with `wr_way = 0`.

**I_ST_LABEL** (*reg_val, reg_addr, offset_, label_*)

Store value from register `reg_val` with label to lower 16 bits of RTC memory address.

This instruction is equivalent to calling I_ST_MANUAL() instruction with `label = label_`, `upper = 0` and `wr_way = 1`.

---

```c
|    |    | | Write full-word, including |
|    |    | | | RTC_SLOW_MEM[addr + offset_]{31:0} = {insn_PC[10:0], 3'b0, label[1:0], reg_val[15:0]} | the PC and the data |
|    |    | -------------------------------|----------------------------|
|    |    | Store the data with label |
|    |    | | | RTC_SLOW_MEM[addr + offset_]{15:0} = {label[1:0], reg_val[13:0]} | in the low half-word |
|    |    | -------------------------------|----------------------------|
|    |    | Store the data with label |
|    |    | | | RTC_SLOW_MEM[addr + offset_]{31:16} = {label[1:0], reg_val[13:0]} | in the high half-word |
|    |    | -------------------------------|----------------------------|
|    |    | Store the data without |
|    |    | | | RTC_SLOW_MEM[addr + offset_]{15:0} = reg_val[15:0] | label in the low half-word |
|    |    | -------------------------------|----------------------------|
|    |    | Store the data without |
|    |    | | | RTC_SLOW_MEM[addr + offset_]{31:16} = reg_val[15:0] | label in the high half-word |
|    |    | -------------------------------|----------------------------|
```
**I_TH_LABEL** (reg_val, reg_addr, offset_, label_)

Store value from register reg_val with label to upper 16 bits of RTC memory address.

This instruction is equivalent to calling `I_TH_MANUAL()` instruction with label = label_, upper = 1 and wr_way = 1.

**I_TH_AUTO** (reg_val, reg_addr, label_, wr_way_)

Store lower half-word, upper half-word or full-word data from register reg_val into RTC memory address with auto-increment of the offset value.

This instruction can be used to write data to continuous addresses in the RTC_SLOW_MEM. The initial address must be set using the SUB_OPCODE_ST_OFFSET instruction before the auto store instruction is called. The data written to the RTC memory address could be written to the full 32 bit word or to the lower half-word or the upper half-word. The storage method is dictated by the wr_way field and the number of times the SUB_OPCODE_ST_AUTO instruction is called. write_cnt indicates the later. The following table summarizes the storage method:

<table>
<thead>
<tr>
<th>wr_way</th>
<th>write_cnt</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Write full-word, including the PC and the data</td>
</tr>
<tr>
<td>0</td>
<td>odd</td>
<td>Store the data with label in the low half-word</td>
</tr>
<tr>
<td>0</td>
<td>even</td>
<td>Store the data without label in the low half-word</td>
</tr>
<tr>
<td>1</td>
<td>odd</td>
<td>Store the data with label in the high half-word</td>
</tr>
<tr>
<td>1</td>
<td>even</td>
<td>Store the data without label in the high half-word</td>
</tr>
</tbody>
</table>

The initial address offset is incremented after each store operation as follows:

- When a full-word is written, the offset is automatically incremented by 1 after each SUB_OPCODE_ST_AUTO operation.
- When a half-word is written (lower half-word first), the offset is automatically incremented by 1 after two SUB_OPCODE_ST_AUTO operations.

SUB_OPCODE_ST_AUTO = manual_en:0, offset_set:0, wr_auto:1
**I_STO** (offset_)

Set the initial address offset for auto-store operation

This instruction sets the initial address of the RTC_SLOW_MEM to be used by the auto-store operation. The offset is incremented automatically. Refer I_ST_AUTO() for detailed explanation.

\[
\text{SUB_OPCODE_ST_OFFSET} = \text{manual_en:0, offset_set:1, wr_auto:1}
\]

**I_STI** (reg_val, reg_addr)

Store value from register reg_val to 32 bit word of the RTC memory address.

This instruction is equivalent to calling I_ST_AUTO() instruction with \( \text{label} = 0 \) and \( \text{wr_way} = 3 \). The data in reg_val will be either written to the lower half-word or the upper half-word of the RTC memory address depending on the count of the number of times the I_STI() instruction is called. The initial offset is automatically incremented with I_STI() is called twice. Refer I_ST_AUTO() for detailed explanation.

**I_STI_LABEL** (reg_val, reg_addr, label_)

Store value from register reg_val with label to 32 bit word of the RTC memory address.

This instruction is equivalent to calling I_ST_AUTO() instruction with \( \text{label} = \text{label}_\text{} \) and \( \text{wr_way} = 1 \). The data in reg_val will be either written to the lower half-word or the upper half-word of the RTC memory address depending on the count of the number of times the I_STI_LABEL() instruction is called. The initial offset is automatically incremented with I_STI_LABEL() is called twice. Refer I_ST_AUTO() for detailed explanation.

**I_STI32** (reg_val, reg_addr, label_)

Store value from register reg_val to full 32 bit word of the RTC memory address.

This instruction is equivalent to calling I_ST_AUTO() instruction with \( \text{label} = \text{label}_\text{} \) and \( \text{wr_way} = 0 \). The data in reg_val will be written to the RTC memory address along with the label and the PC. The initial offset is automatically incremented each time the I_STI32() instruction is called. Refer I_ST_AUTO() for detailed explanation.

**I_LD_MANUAL** (reg_dest, reg_addr, offset_, rd_upper_)

Load lower half-word, upper half-word or full-word data from RTC memory address into the register reg_dest.

This instruction reads the lower half-word or upper half-word of the RTC memory address depending on the value of rd_upper_. The following table summarizes the loading method:

<table>
<thead>
<tr>
<th>rd_upper</th>
<th>data</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reg_dest(15:0) = RTC_SLOW_MEM<a href="">addr + offset_</a></td>
<td>Read...</td>
</tr>
<tr>
<td>1</td>
<td>reg_dest(15:0) = RTC_SLOW_MEM<a href="">addr + offset_</a></td>
<td>Read...</td>
</tr>
</tbody>
</table>

**I_LD** (reg_dest, reg_addr, offset_)

Load lower 16 bits value from RTC memory into reg_dest register.
Chapter 2. API Reference

Loads 16 LSBs (rd_upper = 1) from RTC memory word given by the sum of value in reg_addr and value of offset_. I_LD() instruction provides backward compatibility for code written for esp32 to be run on esp32s2.

I_LDL (reg_dest, reg_addr, offset_)
Load lower 16 bits value from RTC memory into reg_dest register.
I_LDL() instruction and I_LD() instruction can be used interchangably.

I_LDH (reg_dest, reg_addr, offset_)
Load upper 16 bits value from RTC memory into reg_dest register.
Loads 16 MSBs (rd_upper = 0) from RTC memory word given by the sum of value in reg_addr and value of offset_.

I_BL (pc_offset, imm_value)
Branch relative if R0 register less than the 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_BG (pc_offset, imm_value)
Branch relative if R0 register greater than the 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_BE (pc_offset, imm_value)
Branch relative if R0 register is equal to the 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_BSLE (pc_offset, imm_value)
Branch relative if stage_cnt is less than or equal to the 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
Chapter 2. API Reference

I_BSGE (pc_offset, imm_value)
Branch relative if stage_cnt register is greater than or equal to the 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_BSL (pc_offset, imm_value)
Branch relative if stage_cnt register is less than the 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_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 = src1 + 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

I_STAGE_INC (reg_dest, reg_src, imm_)
Increment stage_cnt register by an immediate: stage_cnt = stage_cnt + imm

I_STAGE_DEC (reg_dest, reg_src, imm_)
Decrement stage_cnt register by an immediate: stage_cnt = stage_cnt - imm

I_STAGE_RST (reg_dest, reg_src, imm_)
Reset stage_cnt register by an immediate: stage_cnt = 0
Chapter 2. API Reference

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_BRANCH (label_num)
Token macro used by M_B and M_BX macros. Not to be used directly.

M_BL (label_num, imm_value)
Macro: branch to label label_num if R0 is less than immediate value.
This macro generates two ulpInsn_t values separated by a comma, and should be used when defining contents of ulpInsn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BG (label_num, imm_value)
Macro: branch to label label_num if R0 is greater than immediate value
This macro generates two ulpInsn_t values separated by a comma, and should be used when defining contents of ulpInsn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

M_BE (label_num, imm_value)
Macro: branch to label label_num if R0 equal to the immediate value
This macro generates two ulpInsn_t values separated by a comma, and should be used when defining contents of ulpInsn_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 ulpInsn_t values separated by a comma, and should be used when defining contents of ulpInsn_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 ulpInsn_t values separated by a comma, and should be used when defining contents of ulpInsn_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 ulpInsn_t values separated by a comma, and should be used when defining contents of ulpInsn_t arrays. First value is not a real instruction; it is a token which is removed by ulp_process_macros_and_load function.

Compiling the ULP Code

To compile the ULP FSM code as part of the component, the following steps must be taken:

1. The ULP FSM 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/.

Note: When registering the component (via idf_component_register), this directory should not be added to the SRC_DIRS argument. The logic behind this is that the ESP-IDF build system will compile files found in SRC_DIRS based on their extensions. For .S files, xtensa-esp32s2-elf-as assembler is used. This is not
desirable for ULP FSM assembly files, so the easiest way to achieve the distinction is by placing ULP FSM assembly files into a separate directory. The ULP FSM assembly source files should also not be added to SRCS for the same reason. See the steps below for how to properly add ULP FSM assembly source files.

2. Call `ulp_embed_binary` from the component CMakeLists.txt after registration. For example:

```cmake
... 
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 FSM 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 FSM 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 the 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 FSM 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 a 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.

### Accessing the ULP FSM Program Variables

Global symbols defined in the ULP FSM program may be used inside the main program.

For example, the ULP FSM 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:

```assembly
.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:

```c
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`:

```c
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:

```c
#include "ulp_app_name.h"
```

```c
void init_ulp_vars() {
    ulp_measurement_count = 64;
}
```

### Starting the ULP FSM Program

To run a ULP FSM 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 the Enable Ultra Low Power (ULP) Coprocessor option must be enabled in menuconfig to work with ULP. To select the type of ULP to be used, the ULP Co-processor type option must be set. To reserve memory for the ULP, the RTC slow memory reserved for coprocessor option must be set to a value big enough 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`):

```c
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)) );
}
```

Once the program is loaded into RTC memory, the application can start it by passing the address of the entry point to the `ulp_run` function:

```c
ESP_ERROR_CHECK( ulp_run(&ulp_entry - RTC_SLOW_MEM) );
```

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 FSM application, this symbol must be marked as `.global`:

```asm
.entry
    // code starts here
```

---

Espressif Systems

Submit Document Feedback

Releasemaster
ESP32-S2 ULP Program Flow

ESP32-S2 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 90 kHz RC oscillator). The number of ticks is set using `RTC_CNTL_ULP_CP_TIMER_1_REG` register.

The application can set ULP timer period values by `ulp_set_wakeup_period()` function.

Once the timer counts the number of ticks set in the selected `RTC_CNTL_ULP_CP_TIMER_1_REG` register, ULP coprocessor powers up and starts 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 powers down, and the timer is started again.

To disable the timer (effectively preventing the ULP program from running again), clear the `RTC_CNTL_ULP_CP_SLP_TIMER_EN` bit in the `RTC_CNTL_ULP_CP_TIMER_REG` register. This can be done both from ULP code and from the main program.

Application Examples

- ULP FSM Coprocessor counts pulses on an IO while main CPU is in Deep-sleep: `system/ulp/ulp_fsm/ulp`
- ULP FSM Coprocessor polls ADC in while main CPU is in Deep-sleep: `system/ulp/ulp_fsm/ulp_adc`

API Reference

Header File

- `components/ulp/ulp_fsm/include/ulp_fsm_common.h`
- This header file can be included with:

  ```c
  #include "ulp_fsm_common.h"
  ```

- This header file is a part of the API provided by the `ulp` component. To declare that your component depends on `ulp`, add the following to your CMakeLists.txt:

  ```cmake
  REQUIRES ulp
  ```
  or
  ```cmake
  PRIV_REQUIRES ulp
  ```

Functions

`esp_err_t ulp_isr_register (intr_handler_t fn, void *arg)`

Register ULP wakeup signal ISR.

**Note**: The ISR routine will only be active if the main CPU is not in deepsleep

**Parameters**

- `fn` - ISR callback function
- `arg` - ISR callback function arguments

**Returns**

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if callback function is NULL
- `ESP_ERR_NO_MEM` if heap memory cannot be allocated for the interrupt

`esp_err_t ulp_isr_deregister (intr_handler_t fn, void *arg)`

Deregister ULP wakeup signal ISR.

**Parameters**
Chapter 2. API Reference

- **fn** - ISR callback function
- **arg** - ISR callback function arguments

**Returns**
- ESP_OK on success
- ESP_ERR_INVALID_ARG if callback function is NULL
- ESP_ERR_INVALID_STATE if a handler matching both callback function and its arguments isn’t registered

```c
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.

**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

**Returns**
- ESP_OK on success
- ESP_ERR_NO_MEM if auxiliary temporary structure cannot be allocated
- one of ESP_ERR_ULP_xxx if program is not valid or cannot be loaded

```c
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):

a. MAGIC, (value 0x00706c75, 4 bytes)
b. TEXT_OFFSET, offset of .text section from binary start (2 bytes)
c. TEXT_SIZE, size of .text section (2 bytes)
d. DATA_SIZE, size of .data section (2 bytes)
e. BSS_SIZE, size of .bss section (2 bytes)
f. (TEXT_OFFSET - 12) bytes of arbitrary data (will not be loaded into RTC memory)
g. .text section
h. .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.

**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

**Returns**
- 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

```c
esp_err_t ulp_run (uint32_t entry_point)
```

Run the program loaded into RTC memory.

**Parameters**
- **entry_point** – entry point, expressed in 32-bit words

**Returns**
- ESP_OK on success

**Macros**

**ESP_ERR_ULP_BASE**

Offset for ULP-related error codes
Chapter 2. API Reference

**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)

### Type Definitions

typedef union ulp_insn ulp_insn_t

### Header File

- components/ulp/ulp_common/include/ulp_common.h
- This header file can be included with:

```c
#include "ulp_common.h"
```

- This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakeLists.txt:

```c
REQUIRES ulp
```
or

```c
PRIV_REQUIRES ulp
```

### Functions

**esp_err_t ulp_set_wakeup_period(size_t period_index, uint32_t period_us)**

Set one of ULP wake up 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 for ESP32, and one period value which can be programmed into RTC_CNTL_ULP_CP_TIMER_1_REG register for ESP32-S2/S3. By default, for ESP32, 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.

For ESP32-S2/S3 the SLEEP instruction not exist. Instead a WAKE instruction will be 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 SM 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.

Parameters

• period_index – wakeup period setting number (0 - 4)
• period_us – wakeup period, us

Returns

• ESP_OK on success
• ESP_ERR_INVALID_ARG if period_index is out of range

void ulp_timer_stop(void)
Stop the ULP timer.

Note: This will stop the ULP from waking up if halted, but will not abort any program currently executing on the ULP.

void ulp_timer_resume (void)
Resume the ULP timer.

Note: This will resume an already configured timer, but does no other configuration

Header File

• components/ulp/ulp_common/include/esp32s2/ulp_common_defs.h
• This header file can be included with:

```
#include "ulp_common_defs.h"
```

• This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakelists.txt:

```
REQUIRES ulp
```

or

```
PRIV_REQUIRES ulp
```

Macros

RTC_SLOW_MEM
RTC slow memory, 8k size

2.9.33 ULP RISC-V Coprocessor Programming

The ULP RISC-V coprocessor is a variant of the ULP present in ESP32-S2. Similar to ULP FSM, the ULP RISC-V coprocessor can perform tasks such as sensor readings while the main CPU stays in low power modes. The main difference between ULP FSM and ULP RISC-V is that the latter can be programmed in C using standard GNU tools. The ULP RISC-V coprocessor can access the RTC_SLOW_MEM memory region, and registers in RTC_CNTL, RTC_IO, and SARADC peripherals. The RISC-V processor is a 32-bit fixed point machine. Its instruction set is based on RV32IMC which includes hardware multiplication and division, and compressed code.
Installing the ULP RISC-V Toolchain

The ULP RISC-V coprocessor code is written in C (assembly is also possible) and compiled using the RISC-V toolchain based on GCC.

If you have already set up ESP-IDF with CMake build system according to the Getting Started Guide, then the toolchain should already be installed.

**Note:** In earlier versions of ESP-IDF, RISC-V toolchain had a different prefix: riscv-none-embed-gcc.

Compiling the ULP RISC-V Code

To compile the ULP RISC-V code as part of the component, the following steps must be taken:

1. The ULP RISC-V code, written in C or assembly (must use the `.S` extension), must be placed in a separate directory inside the component directory, for instance, `ulp/`.

   **Note:** When registering the component (via `idf_component_register`), this directory should not be added to the `SRC_DIRS` argument as it is currently done for the ULP FSM. See the step below for how to properly add ULP source files.

2. Call `ulp_embed_binary` from the component CMakeLists.txt after registration. For example:

   ```
   ...
   idf_component_register()
   
   set(ulp_app_name ulp_${COMPONENT_NAME})
   set(ulp_sources "ulp/ulp_c_source_file.c" "ulp/ulp_assembly_source_file.S")
   set(ulp_exp_dep_srcs "ulp_c_source_file.c")
   ulp_embed_binary(${ulp_app_name} "${ulp_sources}" "${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 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 the 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 source file through the C compiler and assembler.** This step generates the object files 
      (.obj.c or .obj.S depending of source file processed) in the component build directory.
   2. **Run the linker script template through the C preprocessor.** The template is located in components/ulp/ld directory.
   3. **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.
   4. **Dump the contents of the ELF file into a binary** (`ulp_app_name.bin`) which can then be embedded into the application.
   5. **Generate a list of global symbols** (`ulp_app_name.sym`) in the ELF file using `riscv32-esp-elf-nm`.
   6. **Create an LD export script and a header file** (`ulp_app_name.1d` and `ulp_app_name.h`) containing the symbols from `ulp_app_name.sym`. This is done using the `esp32ulp_mapgen.py` utility.
   7. **Add the generated binary to the list of binary files** to be embedded into the application.
Accessing the ULP RISC-V Program Variables

Global symbols defined in the ULP RISC-V program may be used inside the main program.

For example, the ULP RISC-V 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.

```c
volatile int measurement_count;

int some_function()
{
    //read the measurement count for later use.
    int temp = measurement_count;
    ...do something.
}
```

The main program can access the global ULP RISC-V program variables as 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 RISC-V program. Each global symbol defined in the ULP RISC-V program is included in these files and are prefixed with `ulp_`.

The header file contains the declaration of the symbol:

```c
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:

```c
PROVIDE ( ulp_measurement_count = 0x50000060 );
```

To access the ULP RISC-V program variables from the main program, the generated header file should be included using an `include` statement. This will allow the ULP RISC-V program variables to be accessed as regular variables.

```c
#include "ulp_app_name.h"

void init_ulp_vars()
{
    ulp_measurement_count = 64;
}
```

**Mutual Exclusion** If mutual exclusion is needed when accessing a variable shared between the main program and ULP, then this can be achieved by using the ULP RISC-V lock API:

- `ulp_riscv_lock_acquire()`
- `ulp_riscv_lock_release()`

The ULP does not have any hardware instructions to facilitate mutual exclusion, so the lock API achieves this through a software algorithm (Peterson’s algorithm).

The locks are intended to only be called from a single thread in the main program, and will not provide mutual exclusion if used simultaneously from multiple threads.

**Starting the ULP RISC-V Program**

To run a ULP RISC-V program, the main application needs to load the ULP program into RTC memory using the `ulp_riscv_load_binary()` function, and then start it using the `ulp_riscv_run()` function.

Note that the `CONFIG_ULP_COPROC_ENABLED` and `CONFIG_ULP_COPROC_TYPE_RISCV` options must be enabled in menuconfig to work with ULP RISC-V. To reserve memory for the ULP, the `RTC slow memory reserved for coprocessor` option must be set to a value big enough to store ULP RISC-V code and data.
Chapter 2. API Reference

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 RISC-V 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`):

```c
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_riscv_load_binary( bin_start, (bin_end - bin_start)) );
}
```

Once the program is loaded into RTC memory, the application can start it by calling the `ulp_riscv_run()` function:

```c
ESP_ERROR_CHECK( ulp_riscv_run() );
```

**ULP RISC-V Program Flow**

The ULP RISC-V coprocessor is started by a timer. The timer is started once `ulp_riscv_run()` is called. The timer counts the number of RTC_SLOW_CLK ticks (by default, produced by an internal 90 kHz RC oscillator). The number of ticks is set using `RTC_CNTL_ULP_CP_TIMER_1_REG` register. When starting the ULP, `RTC_CNTL_ULP_CP_TIMER_1_REG` will be used to set the number of timer ticks.

The application can set ULP timer period values (`RTC_CNTL_ULP_CP_TIMER_1_REG`) using the `ulp_set_wakeup_period()` function.

Once the timer counts the number of ticks set in the `RTC_CNTL_ULP_CP_TIMER_1_REG` register, the ULP RISC-V coprocessor will power up and start running the program from the entry point set in the call to `ulp_riscv_run()`.

The program runs until the field `RTC_CNTL_COCPU_DONE` in register `RTC_CNTL_COCPU_CTRL_REG` gets written or when a trap occurs due to illegal processor state. Once the program halts, the ULP RISC-V 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_ULP_CP_TIMER_REG` register. This can be done both from the ULP code and from the main program.

**ULP RISC-V Peripheral Support**

To enhance the capabilities of the ULP RISC-V coprocessor, it has access to peripherals which operate in the low-power (RTC) domain. The ULP RISC-V coprocessor can interact with these peripherals when the main CPU is in sleep mode, and can wake up the main CPU once a wakeup condition is reached. The following peripherals are supported.

**RTC I2C** The RTC I2C controller provides I2C master functionality in the RTC domain. The ULP RISC-V coprocessor can read from or write to I2C slave devices using this controller. To use the RTC I2C peripheral, call the `ulp_riscv_i2c_master_init()` from your application running on the main core before initializing the ULP RISC-V core and going to sleep.

Once the RTC I2C controller is initialized, the I2C slave device address must be programmed via the `ulp_riscv_i2c_master_set_slave_addr()` API before any read or write operation is performed.

**Note:** The RTC I2C peripheral always expects a slave sub-register address to be programmed via the `ulp_riscv_i2c_master_set_slave_reg_addr()` API. If it is not, the I2C peripheral uses the `SENS_SAR_I2C_CTRL_REG[18:11]` as the sub-register address for the subsequent read or write operations.
This could make the RTC I2C peripheral incompatible with certain I2C devices or sensors which do not need any sub-register to be programmed.

**Note:** There is no hardware atomicity protection in accessing the RTC I2C peripheral between the main CPU and the ULP RISC-V core. Therefore, care must be taken that both cores are not accessing the peripheral simultaneously.

In case your RTC I2C based ULP RISC-V program is not working as expected, the following sanity checks can help in debugging the issue:

- Incorrect SDA/SCL pin selection: The SDA pin can only be set up as GPIO1 or GPIO3 and SCL pin can only be set up as GPIO0 or GPIO2. Make sure that the pin configuration is correct.
- Incorrect I2C timing parameters: The RTC I2C bus timing configuration is limited by the I2C standard bus specification. Any timing parameters which violate the standard I2C bus specifications would result in an error. For details on the timing parameters, please read the standard I2C specifications.
- If the I2C slave device or sensor does not require a sub-register address to be programmed, it may not be compatible with the RTC I2C peripheral. Please refer the notes above.
- If the RTC driver reports a *Write Failed!* or *Read Failed!* error log when running on the main CPU, then make sure:
  - The I2C slave device or sensor works correctly with the standard I2C master on Espressif SoCs. This would rule out any problems with the I2C slave device itself.
  - If the RTC I2C interrupt status log reports a *TIMEOUT* error or *ACK* error, it could typically mean that the I2C device did not respond to a *START* condition sent out by the RTC I2C controller. This could happen if the I2C slave device is not connected properly to the controller pins or if the I2C slave device is in a bad state. Make sure that the I2C slave device is in a good state and connected properly before continuing.
  - If the RTC I2C interrupt log does not report any error status, it could mean that the driver is not fast enough in receiving data from the I2C slave device. This could happen as the RTC I2C controller does not have a TX/RX FIFO to store multiple bytes of data but rather, it depends on single byte transmissions using an interrupt status polling mechanism. This could be mitigated to some extent by making sure that the SCL clock of the peripheral is running as fast as possible. This can be tweaked by configuring the SCL low period and SCL high period values in the initialization config parameters for the peripheral.
- Other methods of debugging problems would be to ensure that the RTC I2C controller is operational only on the main CPU without any ULP RISC-V code interfering and without any sleep mode being activated. This is the basic configuration under which the RTC I2C peripheral must work. This way you can rule out any potential issues due to the ULP or sleep modes.

### Debugging Your ULP RISC-V Program

When programming the ULP RISC-V, it can sometimes be challenging to figure out why the program is not behaving as expected. Due to the simplicity of the core, many of the standard methods of debugging, e.g., JTAG or `printf`, are simply not available.

Keeping this in mind, here are some ways that may help you debug you ULP RISC-V program:

- Share program state through shared variables: as described in [Accessing the ULP RISC-V Program Variables](#), both the main CPU and the ULP core can easily access global variables in RTC memory. Writing state information to such a variable from the ULP and reading it from the main CPU can help you discern what is happening on the ULP core. The downside of this approach is that it requires the main CPU to be awake, which will not always be the case. Keeping the main CPU awake might even, in some cases, mask problems, as some issues may only occur when certain power domains are powered down.
- Use the bit-banged UART driver to print: the ULP RISC-V component comes with a low-speed bit-banged UART TX driver that can be used for printing information independently of the main CPU state. See `system/ulp/ulp_riscv/uart_print` for an example of how to use this driver.
- Trap signal: the ULP RISC-V has a hardware trap that will trigger under certain conditions, e.g., illegal instruction. This will cause the main CPU to be woken up with the wake-up cause `ESP_SLEEP_WAKEUP_COCPU_TRAP_TRIG`. 

---

*Espressif Systems 1775  Release master*
Application Examples

• ULP RISC-V Coprocessor polls GPIO while main CPU is in deep sleep: system/ulp/ulp_riscv/gpio.
• ULP RISC-V Coprocessor uses bit-banged UART driver to print: system/ulp/ulp_riscv/uart_print.
• ULP RISC-V Coprocessor reads external temperature sensor while main CPU is in deep sleep: system/ulp/ulp_riscv/ds18b20_ow.
• ULP RISC-V Coprocessor reads external I2C temperature and humidity sensor (BMP180) while the main CPU is in Deep-sleep and wakes up the main CPU once a threshold is met: system/ulp/ulp_riscv/i2c.

API Reference

Header File

• components/ulp/ulp_riscv/include/ulp_riscv.h
• This header file can be included with:

```c
#include "ulp_riscv.h"
```

• This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakeLists.txt:

```cmake
REQUIRES ulp
```

or

```cmake
PRIV_REQUIRES ulp
```

Functions

**esp_err_t ulp_riscv_isr_register** *(intr_handler_t fn, void *arg, uint32_t mask)*

Register ULP signal ISR.

**Note:** The ISR routine will only be active if the main CPU is not in deepsleep

**Parameters**

• fn - ISR callback function
• arg - ISR callback function arguments
• mask - Bit mask to enable the required ULP RISC-V interrupt signals

**Returns**

• ESP_OK on success
• ESP_ERR_INVALID_ARG if callback function is NULL or if the interrupt bits are invalid
• ESP_ERR_NO_MEM if heap memory cannot be allocated for the interrupt
• other errors returned by esp_intr_alloc

**esp_err_t ulp_riscv_isr_deregister** *(intr_handler_t fn, void *arg, uint32_t mask)*

Deregister ULP signal ISR.

**Parameters**

• fn - ISR callback function
• arg - ISR callback function arguments
• mask - Bit mask to enable the required ULP RISC-V interrupt signals

**Returns**

• ESP_OK on success
• ESP_ERR_INVALID_ARG if callback function is NULL or if the interrupt bits are invalid
• ESP_ERR_INVALID_STATE if a handler matching both callback function and its arguments isn’t registered
Chapter 2. API Reference

```c
esp_err_t ulp_riscv_config_and_run (ulp_riscv_cfg_t *cfg)
Configure the ULP and run the program loaded into RTC memory.

Parameters cfg - pointer to the config struct

Returns ESP_OK on success
```

```c
esp_err_t ulp_riscv_run (void)
Configure the ULP with default settings and run the program loaded into RTC memory.

Returns ESP_OK on success
```

```c
esp_err_t ulp_riscv_load_binary (const uint8_t *program_binary, size_t program_size_bytes)
Load ULP-RISC-V program binary into RTC memory.
Different than ULP FSM, the binary program has no special format, it is the ELF file generated by RISC-V
toolchain converted to binary format using objcopy.
Linker script in components/ulp/ld/ulp_riscv.ld produces ELF files which correspond to this format. This linker
script produces binaries with load_addr == 0.

Parameters
- program_binary - pointer to program binary
- program_size_bytes - size of the program binary

Returns
- ESP_OK on success
- ESP_ERR_INVALID_SIZE if program_size_bytes is more than 8KiB
```

```c
void ulp_riscv_timer_stop (void)
Stop the ULP timer.

Note: This will stop the ULP from waking up if halted, but will not abort any program currently executing
on the ULP.
```

```c
void ulp_riscv_timer_resume (void)
Resumes the ULP timer.

Note: This will resume an already configured timer, but does no other configuration
```

```c
void ulp_riscv_halt (void)
Halts the program currently running on the ULP-RISC-V.

Note: Program will restart at the next ULP timer trigger if timer is still running. If you want to stop the ULP
from waking up then call ulp_riscv_timer_stop() first.
```

```c
void ulp_riscv_reset (void)
Resets the ULP-RISC-V core from the main CPU.

Note: This will reset the ULP core from the main CPU. It is intended to be used when the ULP is in a bad
state and cannot run as intended due to a corrupt firmware or any other reason. The main core can reset the
ULP core with this API and then re-initialize the ULP.
```

**Structures**

```c
struct ulp_riscv_cfg_t
ULP riscv init parameters.
```
Chapter 2. API Reference

Public Members

\texttt{ulp_riscv_wakeup_source_t} \texttt{wakeup\_source}

ULP wakeup source

 Macros

\texttt{ULP\_RISCV\_DEFAULT\_CONFIG()}

\texttt{ULP\_RISCV\_SW\_INT}

\texttt{ULP\_RISCV\_TRAP\_INT}

 Enumerations

\texttt{enum ulp_riscv_wakeup_source_t}

Values:

- \texttt{ULP\_RISCV\_WAKEUP\_SOURCE\_TIMER}
- \texttt{ULP\_RISCV\_WAKEUP\_SOURCE\_GPIO}

 Header File

- \texttt{components/ulp/ulp_riscv/shared/include/ulp_riscv_lock_shared.h}
- This header file can be included with:

\begin{verbatim}
#include "ulp_riscv_lock_shared.h"
\end{verbatim}

- This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakeLists.txt:

\begin{verbatim}
REQUIRES ulp
\end{verbatim}

or

\begin{verbatim}
PRIV_REQUIRES ulp
\end{verbatim}

 Structures

\texttt{struct ulp_riscv_lock_t}

Structure representing a lock shared between ULP and main CPU.

 Public Members

\begin{verbatim}
bool critical_section_flag_ulp
ULP wants to enter the critical sections
\end{verbatim}

\begin{verbatim}
bool critical_section_flag_main_cpu
Main CPU wants to enter the critical sections
\end{verbatim}

\begin{verbatim}
ulp_riscv_lock_turn_t turn
Which CPU is allowed to enter the critical section
\end{verbatim}
Enumerations

enum ulp_riscv_lock_turn_t
Enum representing which processor is allowed to enter the critical section.

Values:

enumerator ULP_RISCV_LOCK_TURN_UPLP
ULP’s turn to enter the critical section

enumerator ULP_RISCV_LOCK_TURN_MAIN_CPU
Main CPU’s turn to enter the critical section

Header File

- components/ulp/ulp_riscv/include/ulp_riscv_lock.h
- This header file can be included with:
  ```
  #include "ulp_riscv_lock.h"
  ```
- This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakeLists.txt:
  ```
  REQUIRES ulp
  ```
  or
  ```
  PRIV_REQUIRES ulp
  ```

Functions

void ulp_riscv_lock_acquire (ulp_riscv_lock_t *lock)
Locks are based on the Peterson’s algorithm, [https://en.wikipedia.org/wiki/Peterson%27s_algorithm](https://en.wikipedia.org/wiki/Peterson%27s_algorithm).
Acquire the lock, preventing the ULP from taking until released. Spins until lock is acquired.

**Note:** The lock is only designed for being used by a single thread on the main CPU, it is not safe to try to acquire it from multiple threads.

Parameters

lock – Pointer to lock struct, shared with ULP

void ulp_riscv_lock_release (ulp_riscv_lock_t *lock)
Release the lock.

Parameters

lock – Pointer to lock struct, shared with ULP

Header File

- components/ulp/ulp_riscv/include/ulp_riscv_i2c.h
- This header file can be included with:
  ```
  #include "ulp_riscv_i2c.h"
  ```
- This header file is a part of the API provided by the ulp component. To declare that your component depends on ulp, add the following to your CMakeLists.txt:
  ```
  REQUIRES ulp
  ```
  or
Chapter 2. API Reference

Functions

void *ulp_riscv_i2c_master_set_slave_addr (uint8_t slave_addr)
Set the I2C slave device address.

Parameters
slave_addr – I2C slave address (7 bit)

void *ulp_riscv_i2c_master_set_slave_reg_addr (uint8_t slave_reg_addr)
Set the I2C slave device sub register address.

Note: The RTC I2C peripheral always expects a slave sub register address to be programmed. If it is not, the I2C peripheral uses the SENS_SAR_I2C_CTRL_REG[18:11] as the sub register address for the subsequent read or write operation.

Parameters
slave_reg_addr – I2C slave sub register address

void *ulp_riscv_i2c_master_read_from_device (uint8_t *data_rd, size_t size)
Read from I2C slave device.

Note: The I2C slave device address must be configured at least once before invoking this API.

Parameters
• data_rd – Buffer to hold data to be read
• size – Size of data to be read in bytes

void *ulp_riscv_i2c_master_write_to_device (uint8_t *data_wr, size_t size)
Write to I2C slave device.

Note: The I2C slave device address must be configured at least once before invoking this API.

Parameters
• data_wr – Buffer which holds the data to be written
• size – Size of data to be written in bytes

esp_err_t *ulp_riscv_i2c_master_init (const ulp_riscv_i2c_cfg_t *cfg)
Initialize and configure the RTC I2C for use by ULP RISC-V. Currently RTC I2C can only be used in master mode.

Parameters
• cfg - Configuration parameters

Returns
esp_err_t ESP_OK when successful

Structures

struct ulp_riscv_i2c_pin_cfg_t
ULP RISC-V RTC I2C pin config.

Public Members
## Chapter 2. API Reference

### `uint32_t sda_io_num`

GPIO pin for SDA signal. Only GPIO#1 or GPIO#3 can be used as the SDA pin.

### `uint32_t scl_io_num`

GPIO pin for SCL signal. Only GPIO#0 or GPIO#2 can be used as the SCL pin.

### `bool sda_pullup_en`

SDA line enable internal pullup. Can be configured if external pullup is not used.

### `bool scl_pullup_en`

SCL line enable internal pullup. Can be configured if external pullup is not used.

### `struct ulp_riscv_i2c_timing_cfg_t`

ULP RISC-V RTC I2C timing config.

#### Public Members

- **`float scl_low_period`**
  
  SCL low period in micro seconds

- **`float scl_high_period`**
  
  SCL high period in micro seconds

- **`float sda_duty_period`**
  
  Period between the SDA switch and the falling edge of SCL in micro seconds

- **`float scl_start_period`**
  
  Waiting time after the START condition in micro seconds

- **`float scl_stop_period`**
  
  Waiting time before the END condition in micro seconds

- **`float i2c_trans_timeout`**
  
  I2C transaction timeout in micro seconds

### `struct ulp_riscv_i2c_cfg_t`

ULP RISC-V RTC I2C init parameters.

#### Public Members

- **`ulp_riscv_i2c_pin_cfg_t i2c_pin_cfg`**
  
  RTC I2C pin configuration

- **`ulp_riscv_i2c_timing_cfg_t i2c_timing_cfg`**
  
  RTC I2C timing configuration
2.9.34 Watchdogs

Overview

ESP-IDF supports multiple types of watchdogs:

- Interrupt Watchdog Timer (IWDT)
- Task Watchdog Timer (TWDT)
- XTAL32K Watchdog Timer (Crystal 32K Watchdog Timer, i.e., XTWDT)

The Interrupt Watchdog is responsible for ensuring that ISRs (Interrupt Service Routines) are not blocked for a prolonged period of time. The TWDT is responsible for detecting instances of tasks running without yielding for a prolonged period.

The various watchdog timers can be enabled using the Project Configuration Menu. However, the TWDT can also be enabled during runtime.

Interrupt Watchdog Timer (IWDT)

The purpose of the IWDT is to ensure that interrupt service routines (ISRs) are not blocked from running for a prolonged period of time (i.e., the IWDT timeout period). Preventing ISRs from running in a timely manner is undesirable as it can increase ISR latency, and also prevent task switching (as task switching is executed form an ISR). The things that can block ISRs from running include:

- Disabling interrupts
- Critical Sections (also disables interrupts)
- Other same/higher priority ISRs which block same/lower priority ISRs from running

The IWDT utilizes the watchdog timer in Timer Group 1 as its underlying hardware timer and leverages the FreeRTOS tick interrupt on each CPU to feed the watchdog timer. If the tick interrupt on a particular CPU is not run at within the IWDT timeout period, it is indicative that something is blocking ISRs from being run on that CPU (see the list of reasons above).

When the IWDT times out, the default action is to invoke the panic handler and display the panic reason as Interrupt wdt timeout on CPU0 or Interrupt wdt timeout on CPU1 (as applicable). Depending on the panic handler’s configured behavior (see CONFIG_ESP_SYSTEM_PANIC), users can then debug the source of the IWDT timeout (via the backtrace, OpenOCD, gdbstub etc) or simply reset the chip (which may be preferred in a production environment).

If for whatever reason the panic handler is unable to run after an IWDT timeout, the IWDT has a second stage timeout that will hard-reset the chip (i.e., a system reset).

Configuration

- The IWDT is enabled by default via the CONFIG_ESP_INT_WDT option.
- The IWDT’s timeout is configured by setting the CONFIG_ESP_INT_WDT_TIMEOUT_MS option.
  - Note that the default timeout is higher if PSRAM support is enabled, as a critical section or interrupt routine that accesses a large amount of PSRAM takes longer to complete in some circumstances.
The timeout should always at least twice longer than the period between FreeRTOS ticks (see CONFIG_FREERTOS_HZ).

**Tuning**  If you find the IWDT timeout is triggered because an interrupt or critical section is running longer than the timeout period, consider rewriting the code:

- Critical sections should be made as short as possible. Any non-critical code/computation should be placed outside the critical section.
- Interrupt handlers should also perform the minimum possible amount of computation. Users can consider deferring any computation to a task by having the ISR push data to a task using queues.

Neither critical sections or interrupt handlers should ever block waiting for another event to occur. If changing the code to reduce the processing time is not possible or desirable, it is possible to increase the CONFIG_ESP_INT_WDT_TIMEOUT_MS setting instead.

**Task Watchdog Timer (TWDT)**

The Task Watchdog Timer (TWDT) is used to monitor particular tasks, ensuring that they are able to execute within a given timeout period. The TWDT primarily watches the Idle Tasks of each CPU, however any task can subscribe to be watched by the TWDT. By watching the Idle Tasks of each CPU, the TWDT can detect instances of tasks running for a prolonged period of time without yielding. This can be an indicator of poorly written code that spinloops on a peripheral, or a task that is stuck in an infinite loop.

The TWDT is built around the Hardware Watchdog Timer in Timer Group 0. When a timeout occurs, an interrupt is triggered.

Users can define the function esp_task_wdt_isr_user_handler in the user code, in order to receive the timeout event and extend the default behavior.

**Usage**  The following functions can be used to watch tasks using the TWDT:

- esp_task_wdt_init() to initialize the TWDT and subscribe the idle tasks.
- esp_task_wdt_add() subscribes other tasks to the TWDT.
- Once subscribed, esp_task_wdt_reset() should be called from the task to feed the TWDT.
- esp_task_wdt_delete() unsubscribes a previously subscribed task.
- esp_task_wdt_deinit() unsubscribes the idle tasks and deinitializes the TWDT.

In the case where applications need to watch at a more granular level (i.e., ensure that a particular functions/stub/code-path is called), the TWDT allows subscription of users.

- esp_task_wdt_add_user() to subscribe an arbitrary user of the TWDT. This function returns a user handle to the added user.
- esp_task_wdt_reset_user() must be called using the user handle in order to prevent a TWDT timeout.
- esp_task_wdt_delete_user() unsubscribes an arbitrary user of the TWDT.

**Configuration**  The default timeout period for the TWDT is set using config item CONFIG_ESP_TASK_WDT_TIMEOUT_S. This should be set to at least as long as you expect any single task needs to monopolize the CPU (for example, if you expect the app will do a long intensive calculation and should not yield to other tasks). It is also possible to change this timeout at runtime by calling esp_task_wdt_init().

**Note:**  Erasing large flash areas can be time consuming and can cause a task to run continuously, thus triggering a TWDT timeout. The following two methods can be used to avoid this:

- Increase CONFIG_ESP_TASK_WDT_TIMEOUT_S in menuconfig for a larger watchdog timeout period.
- You can also call esp_task_wdt_init() to increase the watchdog timeout period before erasing a large flash area.
For more information, you can refer to *SPI Flash API*.

The following config options control TWDT configuration. They are all enabled by default:

- `CONFIG_ESP_TASK_WDT_EN` - enables TWDT feature. If this option is disabled, TWDT cannot be used, even if initialized at runtime.
- `CONFIG_ESP_TASK_WDT_INIT` - the TWDT is initialized automatically during startup. If this option is disabled, it is still possible to initialize the Task WDT at runtime by calling `esp_task_wdt_init()`.
- `CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0` - Idle task is subscribed to the TWDT during startup. If this option is disabled, it is still possible to subscribe the idle task by calling `esp_task_wdt_init()` again.

**Note:** On a TWDT timeout the default behaviour is to simply print a warning and a backtrace before continuing running the app. If you want a timeout to cause a panic and a system reset then this can be configured through `CONFIG_ESP_TASK_WDT_PANIC`.

### XTAL32K Watchdog Timer (XTWDT)

One of the optional clock inputs to the ESP32-S2 is an external 32 kHz crystal oscillator (XTAL32K) that is used as a clock source (`XTAL32K_CLK`) to various subsystems (such as the RTC).

The XTWDT is a dedicated watchdog timer used to ensure that the XTAL32K is functioning correctly. When `XTAL32K_CLK` works as the clock source of `RTC_SLOW_CLK` and stops oscillating, the XTWDT will detect this and generate an interrupt. It also provides functionality for automatically switching over to the internal, but less accurate oscillator as the `RTC_SLOW_CLK` source.

Since the switch to the backup clock is done in hardware it can also happen during Deep Sleep. This means that even if `XTAL32K_CLK` stops functioning while the chip is in Deep Sleep, waiting for a timer to expire, it is still able to wake-up as planned.

If the `XTAL32K_CLK` starts functioning normally again, you can call `esp_xt_wdt_restore_clk` to switch back to this clock source and re-enable the watchdog timer.

**Configuration**

- When the external 32 KHz crystal or oscillator is selected (`CONFIG_RTC_CLK_SRC`) the XTWDT can be enabled via the `CONFIG_ESP_XT_WDT` configuration option.
- The timeout is configured by setting the `CONFIG_ESP_XT_WDT_TIMEOUT` option.
- The automatic backup clock functionality is enabled via the ref:`CONFIG_ESP_XT_WDT_BACKUP_CLK_ENABLE` configuration option.

### JTAG & Watchdogs

While debugging using OpenOCD, the CPUs are 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 reenable 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-S2 is connected to OpenOCD via JTAG.

**API Reference**

- **Task Watchdog** A full example using the Task Watchdog is available in esp-idf: `system/task_watchdog`
Header File

- components/esp_system/include/esp_task_wdt.h
- This header file can be included with:

```c
#include "esp_task_wdt.h"
```

Functions

`esp_err_t esp_task_wdt_init (const esp_task_wdt_config_t *config)`

Initialize the Task Watchdog Timer (TWDT)

This function configures and initializes the TWDT. This function will subscribe the idle tasks if configured to do so. For other tasks, users can subscribe them using esp_task_wdt_add() or esp_task_wdt_add_user(). This function won’t start the timer if no task have been registered yet.

**Note:** `esp_task_wdt_init()` must only be called after the scheduler is started. Moreover, it must not be called by multiple tasks simultaneously.

**Parameters**

- `config` - Configuration structure

**Returns**

- ESP_OK: Initialization was successful
- ESP_ERR_INVALID_STATE: Already initialized
- Other: Failed to initialize TWDT

`esp_err_t esp_task_wdt_reconfigure (const esp_task_wdt_config_t *config)`

Reconfigure the Task Watchdog Timer (TWDT)

The function reconfigures the running TWDT. It must already be initialized when this function is called.

**Note:** `esp_task_wdt_reconfigure()` must not be called by multiple tasks simultaneously.

**Parameters**

- `config` - Configuration structure

**Returns**

- ESP_OK: Reconfiguring was successful
- ESP_ERR_INVALID_STATE: TWDT not initialized yet
- Other: Failed to initialize TWDT

`esp_err_t esp_task_wdt_deinit (void)`

Deinitialize the Task Watchdog Timer (TWDT)

This function will deinitialize the TWDT, and unsubscribe any idle tasks. Calling this function whilst other tasks are still subscribed to the TWDT, or when the TWDT is already deinitialized, will result in an error code being returned.

**Note:** `esp_task_wdt_deinit()` must not be called by multiple tasks simultaneously.

**Returns**

- ESP_OK: TWDT successfully deinitialized
- Other: Failed to deinitialize TWDT

`esp_err_t esp_task_wdt_add (TaskHandle_t task_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.
**Parameters** task_handle – Handle of the task. Input NULL to subscribe the current running task to the TWDT

**Returns**
- ESP_OK: Successfully subscribed the task to the TWDT
- Other: Failed to subscribe task

```c
esp_err_t esp_task_wdt_add_user(const char *user_name, esp_task_wdt_user_handle_t *user_handle_ret)
```

Subscribe a user to the Task Watchdog Timer (TWDT)

This function subscribes a user to the TWDT. A user of the TWDT is usually a function that needs to run periodically. Each subscribed user must periodically call `esp_task_wdt_reset_user()` to prevent the TWDT from elapsing its timeout period. Failure to do so will result in a TWDT timeout.

**Parameters**
- user_name – [in] String to identify the user
- user_handle_ret – [out] Handle of the user

**Returns**
- ESP_OK: Successfully subscribed the user to the TWDT
- Other: Failed to subscribe user

```c
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.

**Returns**
- ESP_OK: Successfully reset the TWDT on behalf of the currently running task
- Other: Failed to reset

```c
esp_err_t esp_task_wdt_reset_user(esp_task_wdt_user_handle_t user_handle)
```

Reset the Task Watchdog Timer (TWDT) on behalf of a user.

This function will reset the TWDT on behalf of a user. Each subscribed user must periodically call this function to prevent the TWDT from timing out. If one or more subscribed users fail to reset the TWDT on their own behalf, a TWDT timeout will occur.

**Parameters** user_handle – [in] User handle
- ESP_OK: Successfully reset the TWDT on behalf of the user
- Other: Failed to reset

```c
esp_err_t esp_task_wdt_delete(TaskHandle_t task_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()`.

**Parameters** task_handle – [in] Handle of the task. Input NULL to unsubscribe the current running task.

**Returns**
- ESP_OK: Successfully unsubscribed the task from the TWDT
- Other: Failed to unsubscribe task

```c
esp_err_t esp_task_wdt_delete_user(esp_task_wdt_user_handle_t user_handle)
```

Unsubscribes a user from the Task Watchdog Timer (TWDT)

This function will unsubscribe a user from the TWDT. After being unsubscribed, the user should no longer call `esp_task_wdt_reset_user()`.

**Parameters** user_handle – [in] User handle

**Returns**
- ESP_OK: Successfully unsubscribed the user from the TWDT
Chapter 2. API Reference

- Other: Failed to unsubscribe user

`esp_err_t esp_task_wdt_status(TaskHandle_t task_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.

**Parameters**
- `task_handle` - [in] Handle of the task. Input NULL to query the current running task.

**Returns**:
- ESP_OK: The task is currently subscribed to the TWDT
- ESP_ERR_NOT_FOUND: The task is not subscribed
- ESP_ERR_INVALID_STATE: TWDT was never initialized

```c
void esp_task_wdt_isr_user_handler(void)
```

User ISR callback placeholder.

This function is called by `task_wdt_isr` function (ISR for when TWDT times out). It can be defined in user code to handle TWDT events.

**Note:** It has the same limitations as the interrupt function. Do not use ESP_LOGx functions inside.

### Structures

```
struct esp_task_wdt_config_t
```

Task Watchdog Timer (TWDT) configuration structure.

**Public Members**

- `uint32_t timeout_ms`
  TWDT timeout duration in milliseconds

- `uint32_t idle_core_mask`
  Bitmask of the core whose idle task should be subscribed on initialization where `1 << i` means that core `i`’s idle task will be monitored by the TWDT

- `bool trigger_panic`
  Trigger panic when timeout occurs

### Type Definitions

```c
typedef struct esp_task_wdt_user_handle_s *esp_task_wdt_user_handle_t
```

Task Watchdog Timer (TWDT) user handle.

Code examples for this API section are provided in the system directory of ESP-IDF examples.
Chapter 3

Hardware Reference
Chapter 4

API Guides

4.1 Application Level Tracing Library

4.1.1 Overview

ESP-IDF provides a 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-S2 via JTAG, UART, or USB interfaces with small overhead on program execution. It is possible to use JTAG and UART interfaces simultaneously. The UART interface is mostly used for connection with SEGGER SystemView tool (see SystemView).

Developers can use this library to send application-specific state of execution to the host and receive commands or other types of information from 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.

Tracing components used when working over JTAG interface are shown in the figure below.

4.1.2 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 the host has read all the data from HW UP BUFFER, but directly overwrites old data with the new ones. This mode is useful when only the latest trace data is interesting to the user, e.g., for analyzing program’s behavior just before the crash. The 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 the host connects to ESP32-S2. In this mode, before writing new data to HW UP BUFFER, the tracing module checks that whether 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 the trace buffer using the finite value of the maximum waiting time, it is possible that this data will be dropped. This is especially true for tracing from time critical code (ISRs, OS scheduler code, etc.) where 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. But it will not help when the average bitrate of the trace data stream exceeds the hardware interface capabilities.

### 4.1.3 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. Please go to Component config > Application Level Tracing menu, which allows selecting destination for the trace data (hardware interface for transport: JTAG or/and UART). Choosing any of the destinations automatically enables the CONFIG_APPTRACE_ENABLE option. For UART interfaces, users have to define baud rate, TX and RX pins numbers, and additional UART-related parameters.

**Note:** In order to achieve higher data rates and minimize the number of dropped packets, it is recommended to optimize the setting of JTAG clock frequency, so that 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 this mode, trace data is 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, the trace data is overwritten in 16 KB granularity. On panic, the latest data from the current input block is exposed to the host and the host can read them for post-analysis. System panic may occur when a 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 the host will see the latest, but very small piece of the trace.
Chapter 4. API Guides

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, users can decide that it needs no less than 512 bytes of the recent trace data, so if there is less then 512 bytes of pending data at the moment of panic, they will not be flushed and will not overwrite the 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 it controls the maximum time that the tracing module will wait for the host to read the last data in case of panic.

3. **UART RX/TX ring buffer size** (CONFIG_APPTRACE_UART_TX_BUFF_SIZE). The size of the buffer depends on the amount of data transferred through the UART.

4. **UART TX message size** (CONFIG_APPTRACE_UART_TX_MSG_SIZE). The maximum size of the single message to transfer.

### 4.1.4 How to Use This Library

This library provides APIs for transferring arbitrary data between the host and ESP32-S2. When enabled in menuconfig, the target application tracing module is initialized automatically at the system startup, so all what the user needs to do is to call corresponding APIs to send, receive or flush the data.

#### Application Specific Tracing

In general, users 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 the target and the host:

1. On the target side, users should implement algorithms for writing trace data to the host. Piece of code below shows an example on how to do this.

```c
#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.

```c
#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, the user may want to receive data from the host. Piece of code below shows an example on 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*/);
if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to read data from host!");
    return res;
}
if (sz > 0) {
    /* we have data, process them */
    ...
}
```

```
#include "esp_app_trace.h"
...
char down_buf[32];
uint32_t *number;
size_t sz = 32;

/* config down buffer */
esp_apptrace_down_buffer_config(down_buf, sizeof(down_buf));
char *ptr = (char *)esp_apptrace_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_apptrace_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;
}
```

esp_apptrace_read() function uses memcpy to copy host data to user buffer. In
some cases it can be more optimal to use esp_apptrace_down_buffer_get() and
esp_apptrace_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.

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 the 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 the format of data is defined by users, 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: apptrace_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 the 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 the task/ISR which fills the buffer is preempted by another high priority task/ISR. So it is possible that the user data preparation process is not completed at the moment when that chunk is read by the host. To handle such conditions, the tracing module prepends all user data chunks with header which contains the allocated user buffer size (2 bytes) and the length of the actually written data (2 bytes). So the 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 the contents of the whole user chunk (including unfilled area) to the 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:

esp apptrace [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 the file `trace.log`. The file will be saved in the `openocd-esp32` directory.

```
esp apptrace 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 it is likely that the target is not sending enough data to empty the buffer to OpenOCD before the timeout. Either increase the timeout or use the function `esp_apptrace_flush()` to flush the data on specific intervals.

2. Retrieve tracing data indefinitely in non-blocking mode.

```
esp apptrace start file://trace.log 1 -1 -1 0 0
```

There is no limitation on the size of collected data and there is no data timeout set. This process may be stopped by issuing `esp apptrace stop` command on OpenOCD telnet prompt, or by pressing Ctrl+C in OpenOCD window.

3. Retrieve tracing data and save them indefinitely.

```
esp apptrace 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 the OpenOCD window.

4. Wait for the target to be halted. Then resume the target’s operation and start data retrieval. Stop after collecting 2048 bytes of data:

```
esp apptrace start file://trace.log 0 2048 -1 1 0
```

To configure tracing immediately after reset, use the OpenOCD `reset halt` command.

**Logging to Host**

ESP-IDF implements a useful feature: logging to the 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. Most part of the work is done on the host.

By default, ESP-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 the implementation of the `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. And 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 to move string formatting procedure to the host.

The application level tracing feature can be used to transfer log information to the host using `esp_apptrace_vprintf` function. This function does not perform full parsing of the format string and arguments. Instead, it just calculates the number of arguments passed and sends them along with the format string address to the host. On the host, log data is processed and printed out by a special Python script.

**Limitations**  Current implementation of logging over JTAG has some limitations:
Chapter 4. API Guides

1. No support for tracing from ESP_EARLY_LOGx macros.
2. No support for printf arguments whose size exceeds 4 bytes (e.g., double and uint64_t).
3. Only strings from the .rodata section are supported as format strings and arguments.
4. The maximum number of printf arguments is 256.

How To Use It

In order to use logging via trace module, users need to perform the following steps:

1. On the target side, the special vprintf-like function esp_apptrace_vprintf() needs to be installed. It sends log data to the host. An example is esp_log_set_vprintf(esp_apptrace_vprintf);. To send log data to UART again, use esp_log_set_vprintf(vprintf);.
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 ESP-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. It is possible to view events in real-time through the UART interface.

How To Use It

Support for this feature is enabled by Component config > Application Level Tracing > FreeRTOS SystemView Tracing (CONFIG_APPTTRACE_SV_ENABLE) menuconfig option. There are several other options enabled under the same menu:

1. SystemView destination. Select the destination interface: JTAG or UART. In case of UART, it will be possible to connect SystemView application to the ESP32-S2 directly and receive data in real-time.
2. ESP32-S2 timer to use as SystemView timestamp source: (CONFIG_APPTTRACE_SV_TS_SOURCE) selects the source of timestamps for SystemView events. In the single core mode, timestamps are generated using ESP32-S2 internal cycle counter running at maximum 240 Mhz (about 4 ns granularity). In the dual-core mode, external timer working at 40 Mhz is used, so the timestamp granularity is 25 ns.
3. Individually enabled or disabled collection of SystemView events (CONFIG_APPTTRACE_SV_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
Chapter 4. API Guides

- Timer Exit Event

ESP-IDF has all the code required to produce SystemView compatible traces, so users 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.

4. Select Pro or App CPU in menuconfig options Component config > Application Level Tracing > FreeRTOS SystemView Tracing to trace over the UART interface in real-time.

OpenOCD SystemView Tracing Command Options

Command usage:

```
esp 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 the OpenOCD window (not the 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.

   ```
   esp sysview start file://pro-cpu.SVDat file://app-cpu.SVDat
   ```

   The tracing data will be retrieved and saved in non-blocking mode. To stop this process, enter `esp sysview stop` command on OpenOCD telnet prompt, optionally pressing Ctrl+C in the OpenOCD window.

2. Retrieve tracing data and save them indefinitely.

   ```
   esp 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 the OpenOCD window.

**Data Visualization** After trace data are collected, users can use a special tool to visualize the results and inspect behavior of the program.

It is uneasy and awkward to analyze data for every core in separate instance of the tool. Fortunately, there is an Eclipse plugin called **Impulse** which can load several trace files, thus making 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 the free version of SystemView.

Good instructions on how to install, configure, and visualize data in Impulse from one core can be found [here](#).
**Note:** ESP-IDF uses its own mapping for SystemView FreeRTOS events IDs, so users need to replace the original file mapping $SYSVIEW_INSTALL_DIR/Description/SYSVIEW_FreeRTOS.txt with $IDF_PATH/tools/esp_app_trace/SYSVIEW_FreeRTOS.txt. Also, contents of that ESP-IDF-specific file should be used when configuring SystemView serializer using the above link.

---

**Gcov (Source Code Coverage)**

**Basics of Gcov and Gcovr** 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 Gcovr tool is a utility for managing Gcov and generating summarized code coverage results.

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 Gcovr can be used to 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 Gcovr will generate a coverage report in HTML format.

**Gcov and Gcovr in ESP-IDF** Using Gcov in ESP-IDF is complicated due to 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, you can add `target_compile_options(${COMPONENT_LIB} PRIVATE --coverage)` to the CMakeLists.txt file of the component.
- To cause a select number of source files (e.g., source1.c and source2.c) in the same component to be compiled with the **--coverage** option, you can add `set_source_files_properties(source1.c source2.c PROPERTIES COMPILATION_FLAGS --coverage)` to the CMakeLists.txt file of the component.

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, make sure that the following project configuration options are enabled by running `idf.py menuconfig`.  

---

[Submit Document Feedback]
Chapter 4. API Guides

• Enable the application tracing module by selecting `Trace Memory` for the `CONFIG_APPTRACE_DESTINATION1` option.
• Enable Gcov to the 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 <oocd_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 is 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.gcdac`. 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 lifetime. 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 form 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-S2 gcov OpenOCD` command (via a telnet session). Once called, OpenOCD will immediately preempt the ESP32-S2’s current state and execute a built-in ESP-IDF Gcov debug stub function. The debug stub function will handle the dumping of data to the host. Upon completion, the ESP32-S2 will resume its current state.

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 `esp gcov dump` OpenOCD command (via a telnet session). The `esp gcov dump` command will cause OpenOCD to connect to the ESP32-S2, retrieve the code coverage data, then disconnect from the ESP32-S2, 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 esp 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 esp gcov dump` OpenOCD command.

```
b esp_gcov_dump
commands
mon esp gcov dump
end
```

Note: Note that all OpenOCD commands should be invoked in GDB as: `mon <oocd_command>`.
Chapter 4. API Guides

**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 Gcovr can be used to generate code coverage reports. Gcov is provided along with the Xtensa toolchain, whilst Gcovr may need to be installed separately. For details on how to use Gcov or Gcovr, refer to Gcov documentation and Gcovr documentation.

**Adding Gcov Build Target to Project** To make report generation more convenient, users can define additional build targets in their projects such that the report generation can be done with a single build command.

Add the following lines to the `CMakeLists.txt` file of your project.

```cmake
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 gcovr-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.

### 4.2 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. **Application Startup** executes. At this point the second CPU and RTOS scheduler are started.

This process is explained in detail in the following sections.

#### 4.2.1 First Stage Bootloader

After SoC reset, the CPU will start running immediately to perform initialization. The reset vector code is located in the mask ROM of the ESP32-S2 chip and cannot 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 a custom boot mode (such as UART Download Mode) is requested. If this is the case, this custom loader mode is executed from ROM. Otherwise, proceed with boot as if it was due to software CPU reset. Consult ESP32-S2 datasheet for a description of SoC boot modes and how to execute them.
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.

**Note:** During normal boot modes the RTC watchdog is enabled when this happens, so if the process is interrupted or stalled then the watchdog will reset the SOC automatically and repeat the boot process. This may cause the SoC to strap into a new boot mode, if the strapping GPIOs have changed.

Second stage bootloader binary image is loaded from flash starting at address 0x1000. The 4 kB sector of flash before this address is unused.

### 4.2.2 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. 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 by default at offset 0x8000 (configurable value). See partition tables documentation for more information. The bootloader finds factory and OTA app partitions. If OTA app partitions are found in the partition table, the bootloader consults the `otadata` partition to determine which one should be booted. See Over The Air Updates (OTA) for more information.

For a full description of the configuration options available for the ESP-IDF bootloader, see Bootloader.

For the selected partition, second stage bootloader reads the binary image from flash one segment at a time:

- For segments with load addresses in internal IRAM (Instruction RAM) or DRAM (Data RAM), the contents are copied from flash to the load address.
- For segments which have load addresses in DROM (Data Stored in flash) or IROM (Code Executed from flash) regions, the flash MMU is configured to provide the correct mapping from the flash to the load address.

Once all segments are processed - meaning code is loaded and flash MMU is set up, second stage bootloader verifies the integrity of the application and then jumps to the application entry point found in the binary image header.

### 4.2.3 Application Startup

Application startup covers everything that happens after the app starts executing and before the `app_main` function starts running inside the main task. This is split into three stages:

- Port initialization of hardware and basic C runtime environment.
- System initialization of software services and FreeRTOS.
- Running the main task and calling `app_main`.

**Note:** Understanding all stages of ESP-IDF app initialization is often not necessary. To understand initialization from the application developer’s perspective only, skip forward to Running the Main Task.

### Port Initialization

ESP-IDF application entry point is `call_start_cpu0` function found in `components/esp_system/port/cpu_start.c`. This function is executed by the second stage bootloader, and never returns.

This port-layer initialization function initializes the basic C Runtime Environment ("CRT") and performs initial configuration of the SoC’s internal hardware:
Chapter 4. API Guides

- Reconfigure CPU exceptions for the app (allowing app interrupt handlers to run, and causing Fatal Errors to be handled using the options configured for the app rather than the simpler error handler provided by ROM).
- If the option CONFIG_BOOTLOADER_WDT_ENABLE is not set then the RTC watchdog timer is disabled.
- Initialize internal memory (data & bss).
- Finish configuring the MMU cache.
- Enable PSRAM if configured.
- Set the CPU clocks to the frequencies configured for the project.
- Initialize memory protection if configured.

Once call_start_cpu0 completes running, it calls the “system layer” initialization function start_cpu0 found in components/esp_system/startup.c.

System Initialization

The main system initialization function is start_cpu0. By default, this function is weak-linked to the function start_cpu0_default. This means that it is possible to override this function to add some additional initialization steps.

The primary system initialization stage includes:

- Log information about this application (project name, App Version, etc.) if default log level enables this.
- Initialize the heap allocator (before this point all allocations must be static or on the stack).
- Initialize newlib component syscalls and time functions.
- Configure the brownout detector.
- Setup libc stdin, stdout, and stderr according to the serial console configuration.
- Perform any security-related checks, including burning efuses that should be burned for this configuration (including permanently limiting ROM download modes).
- Initialize SPI flash API support.
- Call global C++ constructors and any C functions marked with __attribute__((constructor)).

Secondary system initialization allows individual components to be initialized. If a component has an initialization function annotated with the ESP_SYSTEM_INIT_FN macro, it will be called as part of secondary initialization. Component initialization functions have priorities assigned to them to ensure the desired initialization order. The priorities are documented in esp_system/system_init_fn.txt and ESP_SYSTEM_INIT_FN definition in source code are checked against this file.

Running the Main Task

After all other components are initialized, the main task is created and the FreeRTOS scheduler starts running.

After doing some more initialization tasks (that require the scheduler to have started), the main task runs the application-provided function app_main in the firmware.

The main task that runs app_main has a fixed RTOS priority (one higher than the minimum) and a configurable stack size.

Unlike normal FreeRTOS tasks (or embedded C main functions), the app_main task is allowed to return. If this happens, The task is cleaned up and the system will continue running with other RTOS tasks scheduled normally. Therefore, it is possible to implement app_main as either a function that creates other application tasks and then returns, or as a main application task itself.

4.3 Bootloader

The ESP-IDF Software Bootloader performs the following functions:

1. Minimal initial configuration of internal modules;
2. Initialize *Flash Encryption* and/or *Secure* features, if configured;
3. Select the application partition to boot, based on the partition table and ota_data (if any);
4. Load this image to RAM (IRAM & DRAM) and transfer management to the image that was just loaded.

Bootloader is located at the address 0x1000 in the flash.

For a full description of the startup process including the ESP-IDF bootloader, see *Application Startup Flow*.

### 4.3.1 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.

#### 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 information from flash and uses this information to load the rest of the *Second Stage Bootloader* from flash. 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, it will reconfigure the flash using values read from the currently selected app binary’s header (and NOT from the *Second Stage Bootloader* header). This allows an OTA update to change the SPI flash settings in use.

#### 4.3.2 Log Level

The default bootloader log level is “Info”. By setting the `CONFIG_BOOTLOADER_LOG_LEVEL` option, it is possible to increase or decrease this level. This log level is separate from the log level used in the app (see *Logging library*).

Reducing bootloader log verbosity can improve the overall project boot time by a small amount.

#### 4.3.3 Factory Reset

Sometimes it is desirable to have a way for the device to fall back to a known-good state, in case of some problem with an update.

To roll back to the original “factory” device configuration and clear any user settings, configure the config item `CONFIG_BOOTLOADER_FACTORY_RESET` in the bootloader.

The factory reset mechanism allows the device to be factory reset in two ways:

- Clear one or more data partitions. The `CONFIG_BOOTLOADER_DATA_FACTORY_RESET` option allows users to specify which data partitions will be erased when the factory reset is executed.

  Users can specify the names of partitions as a comma-delimited list with optional spaces for readability. (Like this: `nvs, phy_init, nvs_custom`).

  Make sure that the names of partitions specified in the option are the same as those found in the partition table. Partitions of type “app” cannot be specified here.
• Boot from “factory” app partition. Enabling the `CONFIG_BOOTLOADER_OTA_DATA_ERASE` option will cause the device to boot from the default “factory” app partition after a factory reset (or if there is no factory app partition in the partition table then the default ota app partition is selected instead). This reset process involves erasing the OTA data partition which holds the currently selected OTA partition slot. The “factory” app partition slot (if it exists) is never updated via OTA, so resetting to this allows reverting to a “known good” firmware application.

Either or both of these configuration options can be enabled independently.

In addition, the following configuration options control the reset condition:

- `CONFIG_BOOTLOADER_NUM_PIN_FACTORY_RESET` - The input GPIO number used to trigger a factory reset. This GPIO must be pulled low or high (configurable) 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 continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.
- `CONFIG_BOOTLOADER_FACTORY_RESET_PIN_LEVEL` - configure whether a factory reset should trigger on a high or low level of the GPIO. If the GPIO has an internal pullup then this is enabled before the pin is sampled, consult the ESP32-S2 datasheet for details on pin internal pullups.

If an application needs to know if the factory reset has occurred, users can call the function `bootloader_common_get_RTC_retain_mem_factory_reset_state()`.

- If the status is read as true, the function will return the status, indicating that the factory reset has occurred. The function then resets the status to false for subsequent factory reset judgement.
- If the status is read as false, the function will return the status, indicating that the factory reset has not occurred, or the memory where this status is stored is invalid.

Note that this feature reserves some RTC FAST memory (the same size as the `CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP` feature).

### 4.3.4 Boot from Test Firmware

It is possible to write a special firmware app for testing in production, and boot this firmware when needed. The project partition table will need a dedicated app partition entry for this testing app, type `app` and subtype `test` (see Partition Tables).

Implementing a dedicated test app firmware requires creating a totally separate ESP-IDF project for the test app (each project in ESP-IDF only builds one app). The test app can be developed and tested independently of the main project, and then integrated at production testing time as a pre-compiled .bin file which is flashed to the address of the main project’s test app partition.

To support this functionality in the main project’s bootloader, set the configuration item `CONFIG_BOOTLOADER_APP_TEST` and configure the following three items:

- `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. This GPIO must be pulled low or high (configurable) on reset to trigger this. Once the GPIO input is released and the device has been rebooted, the default boot sequence will be enabled again to boot the factory partition or any OTA app partition slot.
- `CONFIG_BOOTLOADER_HOLD_TIME_GPIO` - this is the hold time of GPIO for reset/test mode (by default 5 seconds). The GPIO must be held continuously for this period of time after reset before a factory reset or test partition boot (as applicable) is performed.
- `CONFIG_BOOTLOADER_APP_TEST_PIN_LEVEL` - configure whether a test partition boot should trigger on a high or low level of the GPIO. If the GPIO has an internal pull-up, then this is enabled before the pin is sampled. Consult the ESP32-S2 datasheet for details on pin internal pull-ups.

### 4.3.5 Rollback

Rollback and anti-rollback features must be configured in the bootloader as well.
Consult the App Rollback and Anti-rollback sections in the OTA API reference document.

### 4.3.6 Watchdog

By default, the hardware RTC Watchdog timer remains running while the bootloader is running and will automatically reset the chip if no app has successfully started after 9 seconds.

- The timeout period can be adjusted by setting `CONFIG_BOOTLOADER_WDT_TIME_MS` and recompiling the bootloader.
- The app’s behaviour can be adjusted so the RTC Watchdog remains enabled after app startup. The Watchdog would need to be explicitly reset (i.e., fed) by the app to avoid a reset. To do this, set the `CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE` option, modify the app as needed, and then recompile the app.
- The RTC Watchdog can be disabled in the bootloader by disabling the `CONFIG_BOOTLOADER_WDT_ENABLE` setting and recompiling the bootloader. This is not recommended.

### 4.3.7 Bootloader Size

When enabling additional bootloader functions, including Flash Encryption or Secure Boot, and especially if setting a high `CONFIG_BOOTLOADER_LOG_LEVEL` level, then it is important to monitor the bootloader .bin file’s size.

When using the default `CONFIG_PARTITION_TABLE_OFFSET` value 0x8000, the size limit is 0x8000 bytes.

If the bootloader binary is too large, then the bootloader build will fail with an error “Bootloader binary size [...] is too large for partition table offset”. If the bootloader binary is flashed anyhow then the ESP32-S2 will fail to boot - errors will be logged about either invalid partition table or invalid bootloader checksum.

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 `CONFIG_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.)

When Secure Boot V2 is enabled, there is also an absolute binary size limit of 64 KB (0x10000 bytes) (excluding the 4 KB signature), because the bootloader is first loaded into a fixed size buffer for verification.

### 4.3.8 Fast Boot from Deep-Sleep

The bootloader has the `CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP` option which allows the wake-up time from Deep-sleep to be reduced (useful for reducing power consumption). This option is available when `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.

### 4.3.9 Custom Bootloader

The current bootloader implementation allows a project to extend it or modify it. There are two ways of doing it: by implementing hooks or by overriding it. Both ways are presented in custom_bootloader folder in ESP-IDF examples:

- `bootloader_hooks` which presents how to connect some hooks to the bootloader initialization
- `bootloader_override` which presents how to override the bootloader implementation
In the bootloader space, you cannot use the drivers and functions from other components. If necessary, then the required functionality should be placed in the project’s `bootloader_components` directory (note that this will increase its size).

If the bootloader grows too large then it can collide with the partition table, which is flashed at offset 0x8000 by default. Increase the `partition table offset` value to place the partition table later in the flash. This increases the space available for the bootloader.

### 4.4 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.

#### 4.4.1 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a web server that shows the current humidity, there could be:

- The ESP-IDF base libraries (libc, ROM bindings, etc)
- The Wi-Fi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- A web server
- 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 customize the configuration of the project. A single project contains exactly one project configuration.
- An “app” is an executable that 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 that are compiled into static libraries (.a files) and linked to 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. A full list of supported targets in your version of ESP-IDF can be seen by running `idf.py --list-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 ESP-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.
4.4.2 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
- **Ninja** which builds the project
- **esptool.py** for flashing the target.

You can read more about configuring the build system using idf.py [here](#).

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):

```bash
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:

```bash
ninja flash
```

Or:

```bash
make app-flash
```

Available targets are: `flash`, `app-flash` (app only), `bootloader-flash` (bootloader only).

When flashing this way, optionally set the `ESPSPORT` 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:

```bash
ESPPORT=/dev/ttyUSB0 ninja flash
```
Chapter 4. API Guides

**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 Python version 3.8+.

`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.

---

**4.4.3 Example Project**

An example project directory tree might look like this:

```
- myProject/
  - CMakeLists.txt
  - sdkconfig
  - bootloader_components/
    - boot_component/
      - CMakeLists.txt
      - Kconfig
      - src1.c
  - components/
    - component1/
      - CMakeLists.txt
      - Kconfig
      - src1.c
  - component2/
      - CMakeLists.txt
      - Kconfig
      - src1.c
      - include/
          - component2.h
  - main/
    - CMakeLists.txt
```

(continues on next page)
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 the 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 “bootloader_components” directory contains components that need to be compiled and linked inside the bootloader project. A project does not have to contain custom bootloader components of this kind, but it can be useful in case the bootloader needs to be modified to embed new features.
- 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.
- “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. 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”.
- “build” directory is where the 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.

### 4.4.4 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
cmake_minimum_required(VERSION 3.16)
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 3.16)` tells CMake the minimum version that is required to build the project. ESP-IDF is designed to work with CMake 3.16 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 behavior. Look in /tools/cmake/project.cmake for all of the implementation details.

- COMPONENT_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: 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.
- BOOTLOADER_IGNORE_EXTRA_COMPONENT: A list of components, placed in bootloader_components/, that should be ignored by the bootloader compilation. Use this variable if a bootloader component needs to be included conditionally inside the project.

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 specify 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.

Overriding Default Build Specifications

The build sets some global build specifications (compile flags, definitions, etc.) that gets used in compiling all sources from all components.

For example, one of the default build specifications set is the compile option -Wextra. Suppose a user wants to use override this with -Wno-extra, it should be done after project():

```cmake
cmake_minimum_required(VERSION 3.16)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(myProject)

idf_build_set_property(COMPILE_OPTIONS "-Wno-error" APPEND)
```
This ensures that the compile options set by the user won’t be overridden by the default build specifications, since the latter are set inside `project()`.

### 4.4.5 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 (i.e., 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 (`IDF_PATH/components`), then any components in directories specified in `EXTRA_COMPONENT_DIRS`, and finally the project’s components (`PROJECT_DIR/components`). 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.

**Note:** If a component is overridden in an existing project by moving it to a new location, the project will not automatically see the new component path. Run `idf.py reconfigure` (or delete the project build folder) and then build again.

#### 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 to 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 the 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 **CONFIG_APP_PROJECT_VER_FROM_CONFIG** option is set, the value of **CONFIG_APP_PROJECT_VER** will be used.
  - Else, 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 description will be used.
  - Otherwise, **PROJECT_VER** will be “I”.
- **EXTRA_PARTITION_SUBTYPES**: CMake list of extra partition subtypes. Each subtype description is a comma-separated string with type_name, subtype_name, numeric_value format. Components may add new subtypes by appending them to this list.

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:

```cmake
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:
This can be useful if there is upstream code that emits warnings.

**Note:** CMake `set_source_files_properties` command is not applicable when the source files have been populated with help of the SRC_DIRS variable in `idf_component_register`. See *File Globbing & Incremental Builds* for more details.

When using these commands, place them after the call to `idf_component_register` in the component CMakeLists file.

### 4.4.6 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*.

### 4.4.7 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.

### 4.4.8 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

```c
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 the 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 component 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 in 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:

```bash
- autoProject/
  - CMakeLists.txt
  - components/
    - car/
      - CMakeLists.txt
      - car.c
      - car.h
    - engine/
      - CMakeLists.txt
      - engine.c
      - include/
        - engine.h
    - spark_plug/
      - CMakeLists.txt
      - spark_plug.c
      - spark_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:

```c
/* car.h */
#include "engine.h"

#ifndef ENGINE_IS_HYBRID
#define CAR_MODEL "Hybrid"
#endif
```

And `car.c` includes `car.h` as well:

```c
/* car.c */
#include "car.h"
```

This means the `car/CMakeLists.txt` file needs to declare that `car` requires `engine`:

```c
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.h` 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:
Chapter 4. API Guides

/* 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: cxx, newlib, freertos, esp_hw_support, heap, log, soc, hal, esp_rom, esp_common, esp_system, xtensa/riscv.

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.

Circular Dependencies

It’s possible for a project to contain Component A that requires (REQUIRES or PRIV_REQUIRES) Component B, and Component B that requires Component A. This is known as a dependency cycle or a circular dependency. CMake will usually handle circular dependencies automatically by repeating the component library names twice on the linker command line. However, this strategy doesn’t always work, and the build may fail with a linker error about “Undefined reference to”, referencing a symbol defined by one of the components inside the circular dependency. This is particularly likely if there is a large circular dependency, i.e., A > B > C > D > A.

The best solution is to restructure the components to remove the circular dependency. In most cases, a software architecture without circular dependencies has desirable properties of modularity and clean layering and will be more maintainable in the long term. However, removing circular dependencies is not always possible.

To bypass a linker error caused by a circular dependency, the simplest workaround is to increase the CMake LINK_INTERFACE_MULTIPLICITY property of one of the component libraries. This causes CMake to repeat this library and its dependencies more than twice on the linker command line.

For example:

```cmake
set_property(TARGET ${COMPONENT_LIB} APPEND PROPERTY LINK_INTERFACE_MULTIPLICITY 3)
```

- This line should be placed after idf_component_register in the component CMakeLists.txt file.
- If possible, place this line in the component that creates the circular dependency by depending on a lot of other components. However, the line can be placed inside any component that is part of the cycle. Choosing the component that owns the source file shown in the linker error message, or the component that defines the symbol(s) mentioned in the linker error message, is a good place to start.
- Usually increasing the value to 3 (default is 2) is enough, but if this doesn’t work, then try increasing the number further.
- Adding this option will make the linker command line longer, and the linking stage slower.

Advanced Workaround: Undefined Symbols  
If only one or two symbols are causing a circular dependency, and all other dependencies are linear, then there is an alternative method to avoid linker errors: Specify the specific symbols required for the “reverse” dependency as undefined symbols at link time.

For example, if component A depends on component B but component B also needs to reference reverse_ops from component A (but nothing else), then you can add a line like the following to the component B CMakeLists.txt to resolve the cycle at link time:

```cmake
# This symbol is provided by 'Component A' at link time
target_link_libraries(${COMPONENT_LIB} INTERFACE "-u reverse_ops")
```
The \texttt{-u} argument means that the linker will always include this symbol in the link, regardless of dependency ordering.

This line should be placed after \texttt{idf_component_register} in the component CMakeLists.txt file.

If ‘Component B’ doesn’t need to access any headers of ‘Component A’, only link to a few symbol(s), then this line can be used instead of any \texttt{REQUIRES} from B to A. This further simplifies the component structure in the build system.

See the \texttt{target_link_libraries} documentation for more information about this CMake function.

### Requirements in the Build System Implementation

- Very early in the CMake configuration process, the script \texttt{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 \textit{graph may have cycles}). The graph is used to generate a file \texttt{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 \texttt{BUILD_COMPONENTS} variable). The \texttt{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 \texttt{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 \texttt{BUILD_COMPONENTS} variable determines other orderings during the build:

- Order that \texttt{Project_include.cmake} files are included in the project.
- Order that the list of header paths is generated for compilation (via \texttt{-I} argument). (Note that for a given component’s source files, only that component’s dependency’s header paths are passed to the compiler.)

### Adding Link-Time Dependencies

The ESP-IDF CMake helper function \texttt{idf_component_add_link_dependency} adds a link-only dependency between one component and another. In almost all cases, it is better to use the \texttt{PRIV_REQUIRES} feature in \texttt{idf_component_register} to create a dependency. However, in some cases, it’s necessary to add the link-time dependency of another component to this component, i.e., the reverse order to \texttt{PRIV_REQUIRES} (for example: Overriding Default Chip Drivers).

To make another component depend on this component at link time:

```
idf_component_add_link_dependency(FROM other_component)
```

Place this line after the line with \texttt{idf_component_register}.

It’s also possible to specify both components by name:

```
idf_component_add_link_dependency(FROM other_component TO that_component)
```

### 4.4.9 Overriding Parts of the Project

**Project_include.cmake**

For components that have build requirements that must be evaluated before any component CMakeLists files are evaluated, you can create a file called \texttt{project_include.cmake} in the component directory. This CMake file is included when \texttt{project.cmake} is evaluating the entire project.

\texttt{project_include.cmake} files are used inside ESP-IDF, for defining project-wide build features such as \texttt{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 customized 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 in 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.

Wrappers to Redefine or Extend Existing Functions

Thanks to the linker’s wrap feature, it is possible to redefine or extend the behavior of an existing ESP-IDF function. To do so, you will need to provide the following CMake declaration in your project’s CMakeLists.txt file:

```cmake
target_link_libraries(${COMPONENT_LIB} INTERFACE "-Wl,--wrap=function_to_redefine")
```

Where function_to_redefine is the name of the function to redefine or extend. This option will let the linker replace all the calls to function_to_redefine functions in the binary libraries with calls to __wrap_function_to_redefine function. Thus, you must define this new symbol in your application.

The linker will provide a new symbol named __real_function_to_redefine which points to the former implementation of the function to redefine. It can be called from the new implementation, making it an extension of the former one.

This mechanism is shown in the example build_system/wrappers. Check examples/build_system/wrappers/README.md for more details.

Override the Default Bootloader

Thanks to the optional bootloader_components directory present in your ESP-IDF project, it is possible to override the default ESP-IDF bootloader. To do so, a new bootloader_components/main component should be defined, which will make the project directory tree look like the following:

```
  • myProject/
      – CMakeLists.txt
      – sdkconfig
      – bootloader_components/ - main/ - CMakeLists.txt
         • Kconfig
         • my_bootloader.c
      – main/ - CMakeLists.txt
         • app_main.c
      – build/
```
Chapter 4. API Guides

Here the my_bootloader.c file becomes source code for the new bootloader, which means that it will need to perform all the required operations to set up and load the main application from flash.

It is also possible to conditionally replace the bootloader depending on a certain condition, such as the target for example. This can be achieved thanks to the BOOTLOADER_IGNORE_EXTRA_COMPONENT CMake variable. This list can be used to tell the ESP-IDF bootloader project to ignore and not compile the given components present in bootloader_components. For example, if one wants to use the default bootloader for ESP32 target, then myProject/CMakeLists.txt should look like the following:

```cmake
#include(${ENV[IDF_PATH]}/tools/cmake/project.cmake)

if(${IDF_TARGET} STREQUAL "esp32")
    set(BOOTLOADER_IGNORE_EXTRA_COMPONENT "main")
endif()

project(main)
```

It is important to note that this can also be used for any other bootloader components than main. In all cases, the prefix bootloader_component must not be specified.

See custom_bootloader/bootloader_override for an example of overriding the default bootloader.

### 4.4.10 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 included paths.

### 4.4.11 Debugging CMake

For full details about CMake and CMake commands, see the CMake v3.16 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, the function of warnings on undefined variables is disabled.

To enable this function, we can pass the `--warn-uninitialized` flag to CMake or pass the `--cmake-warn-uninitialized` flag to idf.py 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.

Browse the `/tools/cmake/project.cmake` file and supporting functions in `/tools/cmake/` for more details.
4.4.12 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 preset_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:

```bash
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:

```bash
config ENABLE_LCD_OUTPUT
  bool "Enable LCD output."
  help
    Select this if your board has an 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:

```bash
if(CONFIG_ENABLE_LCD_OUTPUT)
  set(srcs lcd-real.c lcd-spi.c)
else()
  set(srcs lcd-dummy.c)
endif()
```

(continues on next page)
# 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 logo.h
    DEPENDS ${COMPONENT_DIR}/logo.bmp
    VERBATIM)

add_custom_target(logo DEPENDS logo.h)
add_dependencies(${COMPONENT_LIB} logo)

set_property(DIRECTORY "${COMPONENT_DIR}" APPEND PROPERTY ADDITIONAL_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_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 a C source.
You can specify argument EMBED_FILES in the component registration, giving space-delimited names of the files to embed:

```c
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:

```c
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:

```c
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:

```c
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 the “main” component of the file_serving example protocols/http_server/file_serving/main/CMakeLists.txt · two files are loaded at build time and linked into the firmware.

It is also possible to embed a generated file:

```c
add_custom_command(OUTPUT my_processed_file.bin COMMAND my_process_file_cmd my_unprocessed_file.bin)
target_add_binary_data(my_target "my_processed_file.bin" BINARY)
```

In the example above, my_processed_file.bin is generated from my_unprocessed_file.bin through some command my_process_file_cmd, then embedded into the target.

To specify a dependence on a target, use the DEPENDS argument:

```c
add_custom_target(my_process COMMAND ...)
target_add_binary_data(my_target "my_embed_file.bin" BINARY DEPENDS my_process)
```

The DEPENDS argument to target_add_binary_data ensures that the target executes first.

**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:

```cmake
# 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
  ${COMPONENT_DIR}/quirc/lib)
set_directory_properties(
  PROPERTIES ADDITIONAL_CLEAN_FILES
  "${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_CLEAN_FILES`. This means make clean will delete this library. (Note that the other object files from the build won’t be deleted.)

**ExternalProject Dependencies and Clean Builds**

CMake has some unusual behavior around external project builds:

- `ADDITIONAL_CLEAN_FILES` only works when “make” or “ninja” is used as the build system. If 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 behavior, 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.

### 4.4.13 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`. File names that are not specified as full paths are resolved relative to current project’s directory.

When specifying multiple files, use a semicolon as the list separator. Files listed first will be applied first. If a particular key is defined in multiple files, the definition in the latter file will override definitions from former files.

Some of the IDF examples include a `sdkconfig.ci` file. This is part of the continuous integration (CI) test framework and is ignored by the normal build process.

#### Target-dependent Sdkconfig Defaults

If and only if an `sdkconfig.defaults` file exists, the build system will also attempt to load defaults from an `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 there are no generic default settings, an empty `sdkconfig.defaults` still needs to be created if the build system should recognize any additional target-dependent `sdkconfig.defaults`.

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. When there are multiple files in `SDKCONFIG_DEFAULTS`, target-specific file will be applied right after the file bringing it in, before all latter files in `SDKCONFIG_DEFAULTS`.

For example, if `SDKCONFIG_DEFAULTS="sdkconfig.defaults;sdkconfig_devkit1"`, and there is a file `sdkconfig.defaults.esp32` in the same folder, then the files will be applied in the following order: (1) `sdkconfig.defaults` (2) `sdkconfig.defaults.esp32` (3) `sdkconfig_devkit1`.

### 4.4.14 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
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.

### 4.4.15 Building the 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`).

### 4.4.16 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`:

```cmake
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 behavior 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.

### 4.4.17 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:

```cmake
# 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)
```
Chapter 4. API Guides

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 `tinyxml` 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:

```cmake
add_library(foo bar.c fizz.cpp buzz.cpp)
if(ESP_PLATFORM)
    # On ESP-IDF, bar.c needs to include esp_flash.h from the spi_flash component
    target_link_libraries(foo PRIVATE idf::spi_flash)
endif()
```

### 4.4.18 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:

```cmake
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`.

### 4.4.19 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 3.16)
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
# 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.

### 4.4.20 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.

```cpp
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`.

```cpp
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:

```cpp
idf_build_get_property(pyhton 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
- `DEPENDENCIES_LOCK` - lock file path used in component manager. The default value is `dependencies.lock` under the project path.
- `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`
- `EXECUTABLE_DIR` - path containing the output executable
- `IDF_COMPONENT_MANAGER` - the component manager is enabled by default, but if this property is set to 0 it was disabled by the `IDF_COMPONENT_MANAGER` environment variable
Chapter 4. API Guides

- **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**

```cmake
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.

```cmake
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.

```cmake
idf_component_register([[[SRCS src1 src2 ...] | [[SRC_DIRS dir1 dir2 ...]] [EXCLUDE_~SRCS src1 src2 ...]])
```

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 (e.g., .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
• REQUIRE - 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
• KCONFIG - override the default Kconfig file
• KCONFIG_PROJBUILD - override the default Kconfig.projbuild file
• WHOLE_ARCHIVE - if specified, the component library is surrounded by --whole-archive, --no-whole-archive when linked. This has the same effect as setting WHOLE_ARCHIVE component property.

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

### Idp-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_OVERRIDEN_DIR - contains the directory of the original component if **this component overrides another 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
Chapter 4. API Guides

- **KCONFIG_PROJBUILD** - component Kconfig.projbuild; set by `idf_build_component`.
- **LDFRAGMENTS** - list of component linker fragment files; set from `idf_component_register` LDFRAGMENTS argument.
- **MANAGED_PRIV_REQUIRES** - list of private component dependencies added by the IDF component manager from dependencies in `idf_component.yml` manifest file.
- **MANAGED_REQUIRES** - list of public component dependencies added by the IDF component manager from dependencies in `idf_component.yml` manifest file.
- **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 value of `idf_component_register` PRIV_REQUIRES argument and dependencies in `idf_component.yml` manifest file.
- **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 value of `idf_component_register` REQUIRES argument and dependencies in `idf_component.yml` manifest file.
- **SRCS** - list of component source files; set from SRCS or SRC_DIRS/EXCLUDE_SRCS argument of `idf_component_register`.
- **WHOLE_ARCHIVE** - if this property is set to **TRUE** (or any boolean “true” CMake value: 1, ON, YES, Y), the component library is surrounded by `-Wl,--whole-archive`, `-Wl,--no-whole-archive` when linked. This can be used to force the linker to include every object file into the executable, even if the object file doesn’t resolve any references from the rest of the application. This is commonly used when a component contains plugins or modules which rely on link-time registration. This property is **FALSE** by default. It can be set to **TRUE** from the component CMakeLists.txt file.

### 4.4.21 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.
Chapter 4. API Guides

- 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.

### 4.4.22 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 `kconfserver` is provided to allow IDEs to easily integrate with the configuration system logic. `kconfserver` 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 `kconfserver` from a project via `idf.py confserver` or `ninja kconfserver`, or a similar target triggered from a different build generator.

For more information about `kconfserver`, see the [esp-idf-kconfig documentation](#).

### 4.4.23 Build System Internals

#### Build Scripts

The list files 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.
Chapter 4. API Guides

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:

\[\text{Initialization} \rightarrow \text{Enumeration} \rightarrow \text{Processing} \rightarrow \text{Finalization}\]

Fig. 2: 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_COMPONENT_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.

### 4.4.24 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` in ESP-IDF v4.x releases. The script was removed in v5.0 because of its make build system dependency.

**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 `PRIV_INCLUDE_DIRS` 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 `SRCS` 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_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.
Chapter 4. API Guides

- 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 SRCS is also specified (in fact, SRCS 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 Flasing with Ninja or Make for more details.

4.5 C Support

ESP-IDF is primarily written in C and provides C APIs. Newlib is used as standard C library (the Newlib version can be found in newlib/sbom.yml). In general, all C features supported by the compiler, currently GCC, should be available in ESP-IDF, unless specified in Unsupported C Features below.

4.5.1 C Version

GNU dialect of ISO C17 (--std=gnu17) is the current default C version in ESP-IDF.

To compile the source code of a certain component using a different language standard, set the desired compiler flag in the component’s CMakeLists.txt file:

```
idf_component_register( ... )
target_compile_options(${COMPONENT_LIB} PRIVATE -std=gnu11)
```

If the public header files of the component also need to be compiled with the same language standard, replace the flag PRIVATE with PUBLIC.

4.5.2 Unsupported C Features

The following features are not supported in ESP-IDF.
Nested Function Pointers

The GNU dialect of ISO C17 supports nested functions. However, they do not work in ESP-IDF when referenced as function pointer because the compiler generates a trampoline on the stack, while the stack is not executable in ESP-IDF. Hence, do not use function pointers to nested functions.

4.6 C++ Support

ESP-IDF is primarily written in C and provides C APIs. However, ESP-IDF supports development of applications in C++. This document covers various topics relevant to C++ development.

The following C++ features are supported:

- Exception Handling
- Multithreading
- Runtime Type Information (RTTI)
- Thread Local Storage (thread_local keyword)
- All C++ features implemented by GCC, except for some Limitations. See GCC documentation for details on features implemented by GCC.

4.6.1 esp-idf-cxx Component

esp-idf-cxx component provides higher-level C++ APIs for some of the ESP-IDF features. This component is available from the ESP-IDF Component Registry.

4.6.2 C++ Language Standard

By default, ESP-IDF compiles C++ code with C++23 language standard with GNU extensions (-std=gnu++23).

To compile the source code of a certain component using a different language standard, set the desired compiler flag in the component’s CMakeLists.txt file:

```
idf_component_register(...)
target_compile_options(${COMPONENT_LIB} PRIVATE -std=gnu++11)
```

Use PUBLIC instead of PRIVATE if the public header files of the component also need to be compiled with the same language standard.

4.6.3 Multithreading

C++ threads, mutexes, and condition variables are supported. C++ threads are built on top of pthreads, which in turn wrap FreeRTOS tasks.

See cxx/pthread for an example of creating threads in C++.

Note: The destructor of std::jthread can only safely be called from a task that has been created by Thread APIs or by the C++ threading library API.
4.6.4 Exception Handling

Support for C++ Exceptions in ESP-IDF is disabled by default, but can be enabled using the `CONFIG_COMPILER_CXX_EXCEPTIONS` option.

If an exception is thrown, but there is no `catch` block, the program is terminated by the `abort` function, and the backtrace is printed. See Fatal Errors for more information about backtraces.

C++ Exceptions should **only** be used for exceptional cases, i.e., something happening unexpectedly and occurs rarely, such as events that happen less frequently than 1/100 times. **Do not** use them for control flow (see also the section about resource usage below). For more information on how to use C++ Exceptions, see the ISO C++ FAQ and CPP Core Guidelines.

See `cxx/exceptions` for an example of C++ exception handling.

C++ Exception Handling and Resource Usage

Enabling exception handling normally increases application binary size by a few KB.

Additionally, it may be necessary to reserve some amount of RAM for the exception emergency memory pool. Memory from this pool is used if it is not possible to allocate an exception object from the heap.

The amount of memory in the emergency pool can be set using the `CONFIG_COMPILER_CXX_EXCEPTIONS_EMG_POOL_SIZE` variable.

Some additional stack memory (around 200 bytes) is also used if and only if a C++ Exception is actually thrown, because it requires calling some functions from the top of the stack to initiate exception handling.

The run time of code using C++ exceptions depends on what actually happens at run time.

- If no exception is thrown, the code tends to be somewhat faster since there is no need to check error codes.
- If an exception is thrown, the run time of the code that handles exceptions is orders of magnitude slower than code returning an error code.

If an exception is thrown, the run time of the code that unwinds the stack is orders of magnitude slower than code returning an error code. The significance of the increased run time will depend on the application’s requirements and implementation of error handling (e.g., requiring user input or messaging to a cloud). As a result, exception-throwing code should never be used in real-time critical code paths.

4.6.5 Runtime Type Information (RTTI)

Support for RTTI in ESP-IDF is disabled by default, but can be enabled using `CONFIG_COMPILER_CXX_RTTI` option.

Enabling this option compiles all C++ files with RTTI support enabled, which allows using `dynamic_cast` conversion and `typeid` operator. Enabling this option typically increases the binary size by tens of kB.

See `cxx/rtti` for an example of using RTTI in ESP-IDF.

4.6.6 Developing in C++

The following sections provide tips on developing ESP-IDF applications in C++.

Combining C and C++ Code

When an application is developed using both C and C++, it is important to understand the concept of language linkage. In order for a C++ function to be callable from C code, it has to be both **declared** and **defined** with C linkage (extern "C"):
// declaration in the .h file:
#ifdef __cplusplus
extern "C" {
#endif

void my_cpp_func(void);
#endif

// definition in a .cpp file:
extern "C" void my_cpp_func(void) {
  // ...
}

In order for a C function to be callable from C++, it has to be declared with C linkage:

// declaration in .h file:
#ifdef __cplusplus
extern "C" {
#endif

void my_c_func(void);
#endif

// definition in a .c file:
void my_c_func(void) {
  // ...
}

Defining app_main in C++

ESP-IDF expects the application entry point, app_main, to be defined with C linkage. When app_main is defined in a .cpp source file, it has to be designated as extern "C":

extern "C" void app_main()
{
}

Designated Initializers

Many of the ESP-IDF components use Configuration Structures as arguments to the initialization functions. ESP-IDF examples written in C routinely use designated initializers to fill these structures in a readable and maintainable way.

C and C++ languages have different rules with regards to the designated initializers. For example, C++23 (currently the default in ESP-IDF) does not support out-of-order designated initialization, nested designated initialization, mixing of designated initializers and regular initializers, and designated initialization of arrays. Therefore, when porting ESP-IDF C examples to C++, some changes to the structure initializers may be necessary. See the C++ aggregate initialization reference for more details.

iostream

iostream functionality is supported in ESP-IDF, with a couple of caveats:
1. Normally, ESP-IDF build process eliminates the unused code. However, in the case of iostreams, simply including `<iostream>` header in one of the source files significantly increases the binary size by about 200 kB.

2. By default, ESP-IDF uses a simple non-blocking implementation of the standard input stream (stdin). To get the usual behavior of `std::cin`, the application has to initialize the UART driver and enable the blocking mode as shown in `common_components/protocol_examples_common/stdin_out.c`.

### 4.6.7 Limitations

- Linker script generator does not support function level placements for functions with C++ linkage.
- Various section attributes (such as `IRAM_ATTR`) are ignored when used with template functions.
- Vtables are placed into Flash and are not accessible when the flash cache is disabled. Therefore, virtual function calls should be avoided in `IRAM-Safe Interrupt Handlers`. Placement of Vtables cannot be adjusted using the linker script generator, yet.
- C++ filesystem (`std::filesystem`) features are not supported.

### 4.6.8 What to Avoid

Do not use `setjmp/longjmp` in C++. `longjmp` blindly jumps up the stack without calling any destructors, easily introducing undefined behavior and memory leaks. Use C++ exceptions instead, they guarantee correctly calling destructors. If you cannot use C++ exceptions, use alternatives (except `setjmp/longjmp` themselves) such as simple return codes.

## 4.7 Core Dump

### 4.7.1 Overview

A core dump is a set of software state information that is automatically saved by the panic handler when a fatal error occurs. Core dumps are useful for conducting post-mortem analysis of the software’s state at the moment of failure. ESP-IDF provides support for generating core dumps.

A core dump contains snapshots of all tasks in the system at the moment of failure, where each snapshot includes a task’s control block (TCB) and stack. By analyzing the task snapshots, it is possible to find out what task, at what instruction (line of code), and what call stack of that task lead to the crash. It is also possible to dump the contents of variables on demand, provided those variables are assigned special core dump attributes.

Core dump data is saved to a core dump file according to a particular format, see `Core dump internals` for more details. However, ESP-IDF’s `idf.py` command provides special subcommands to decode and analyze the core dump file.

### 4.7.2 Configurations

**Destination**

The `CONFIG_ESP_COREDUMP_TO_FLASH_OR_UART` option enables or disables core dump, and selects the core dump destination if enabled. When a crash occurs, the generated core dump file can either be saved to flash, or output to a connected host over UART.
### Format & Size

The `CONFIG_ESP_COREDUMP_DATA_FORMAT` option controls the format of the core dump file, namely ELF format or Binary format.

The ELF format contains extended features and allows more information regarding erroneous tasks and crashed software to be saved. However, using the ELF format causes the core dump file to be larger. This format is recommended for new software designs and is flexible enough to be extended in future revisions to save more information.

The Binary format is kept for compatibility reasons. Binary format core dump files are smaller while providing better performance.

The `CONFIG_ESP_COREDUMP_MAX_TASKS_NUM` option configures the number of task snapshots saved by the core dump.

Core dump data integrity checking is supported via the `Components > Core dump > Core dump data integrity check` option.

### Reserved Stack Size

Core dump routines run from a separate stack due to core dump itself needing to parse and save all other task stacks. The `CONFIG_ESP_COREDUMP_STACK_SIZE` option controls the size of the core dump’s stack in number of bytes.

Setting this option to 0 bytes will cause the core dump routines to run from the ISR stack, thus saving a bit of memory. Setting the option greater than zero will cause a separate stack to be instantiated.

**Note:** If a separate stack is used, the recommended stack size should be larger than 800 bytes to ensure that the core dump routines themselves do not cause a stack overflow.

### 4.7.3 Core Dump to Flash

When the core dump file is saved to flash, the file is saved to a special core dump partition in flash. Specifying the core dump partition will reserve space on the flash chip to store the core dump file.

The core dump partition is automatically declared when using the default partition table provided by ESP-IDF. However, when using a custom partition table, you need to declare the core dump partition, as illustrated below:

```plaintext
# 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
```

**Important:** If Flash Encryption is enabled on the device, please add an encrypted flag to the core dump partition declaration.

```plaintext
coredump, data, coredump,, 64K, encrypted
```

There are no special requirements for the partition name. It can be chosen according to the application’s needs, but the partition type should be data and the sub-type should be coredump. Also, when choosing partition size, note that the core dump file introduces a constant overhead of 20 bytes and a per-task overhead of 12 bytes. This overhead does not include the size of TCB and stack for every task. So the partition size should be at least 20 + max tasks number x (12 + TCB size + max task stack size) bytes.

An example of the generic command to analyze core dump from flash is:
4.7.4 Core Dump to UART

When the core dump file is output to UART, the output file is Base64-encoded. The `CONFIG_ESP_COREDUMP_DECODE` option allows for selecting whether the output file is automatically decoded by the ESP-IDF monitor or kept encoded for manual decoding.

Automatic Decoding

If `CONFIG_ESP_COREDUMP_DECODE` is set to automatically decode the UART core dump, ESP-IDF monitor will automatically decode the data, translate any function addresses to source code lines, and display it in the monitor. The output to ESP-IDF monitor would resemble the following output:

The `CONFIG_ESP_COREDUMP_UART_DELAY` allows for an optional delay to be added before the core dump file is output to UART.

```
Crashed task handle: 0x3ffc5640, name: 'main', GDB name: 'process 1073501760'

CURRENT THREAD REGISTERS
exccause 0x1d (StoreProhibitedCause)
excvaddr 0x0
epc1 0x40027657
epc2 0x0
...

CURRENT THREAD STACK
#0 0x400251cd in panic_abort (details=0x3ffc553b "abort() was called at PC 0x40087b84 on core 0") at /home/User/esp/esp-idf/components/esp_system/panic.c:452
#1 0x40028970 in esp_system_abort (details=0x3ffc553b "abort() was called at PC 0x40087b84 on core 0") at /home/User/esp/esp-idf/components/esp_system/port/esp_system_chip.c:93
...

THREADS INFO

Id Target Id Frame
1 process 1073501760 0x400251cd in panic_abort (details=0x3ffc553b "abort() was called at PC 0x40087b84 on core 0") at /home/User/esp/esp-idf/components/esp_system/panic.c:452
2 process 1073503644 vPortTaskWrapper (pxCode=0x0, pvParameters=0x0) at /home/User/esp/esp-idf/components/freertos/FreeRTOS-Kernel/portable/xtensa/port.c:161
...

TRACEBACK
#0 0x400251cd in panic_abort (details=0x3ffc553b "abort() was called at PC 0x40087b84 on core 0") at /home/User/esp/esp-idf/components/esp_system/panic.c:452
#1 0x40028970 in esp_system_abort (details=0x3ffc553b "abort() was called at PC 0x40087b84 on core 0") at /home/User/esp/esp-idf/components/esp_system/port/esp_system_chip.c:93
...
```

(continues on next page)
#1 0x40000000 in ?? ()
...
_________________________ ALL MEMORY REGIONS _______________________
Name   Address  Size  Attrs
...  
.iram0.vectors 0x40024000 0x403  R XA
.dram0.data 0x3ffbf1c0 0x2c0c  RW A
...
_________________________ ESP32 CORE DUMP END _______________________

Manual Decoding

If you set `CONFIG_ESP_COREDUMP_DECODE` to no decoding, then the raw Base64-encoded body of core dump is output to UART between the following header and footer of the UART output:

```
================= CORE DUMP START =================
<body of Base64-encoded core dump, save it to file on disk>
================= CORE DUMP END ==================
```

It is advised to manually save the core dump text body to a file. The `CORE DUMP START` and `CORE DUMP END` lines must not be included in a core dump text file. The saved text can be decoded using the following command:

```
idf.py coredump-info -c </path/to/saved/base64/text>
```

or

```
idf.py coredump-debug -c </path/to/saved/base64/text>
```

### 4.7.5 Core Dump Commands

ESP-IDF provides special commands to help to retrieve and analyze core dumps:

- `idf.py coredump-info` - prints crashed task’s registers, call stack, list of available tasks in the system, memory regions, and contents of memory stored in core dump (TCBs and stacks).
- `idf.py coredump-debug` - creates core dump ELF file and runs GDB debug session with this file. You can examine memory, variables, and task states manually. Note that since not all memory is saved in the core dump, only the values of variables allocated on the stack are meaningful.

### 4.7.6 ROM Functions in Backtraces

It is a possible that at the moment of a crash, some tasks and/or the crashed task itself have one or more ROM functions in their call stacks. Since ROM is not part of the program ELF, it is impossible for GDB to parse such call stacks due to GDB analyzing functions’ prologues to decode backtraces. Thus, call stack parsing will break with an error message upon the first ROM function that is encountered.

To overcome this issue, the ROM ELF provided by Espressif is loaded automatically by ESP-IDF monitor based on the target and its revision. More details about ROM ELFs can be found in `esp-rom-elfs`.

### 4.7.7 Dumping Variables on Demand

Sometimes you want to read the last value of a variable to understand the root cause of a crash. Core dump supports retrieving variable data over GDB by applying special attributes to declared variables.
Supported Notations and RAM Regions

- COREDUMP_DRAM_ATTR places the variable into the DRAM area, which is included in the dump.
- COREDUMP_RTC_ATTR places the variable into the RTC area, which is included in the dump.
- COREDUMP_RTC_FAST_ATTR places the variable into the RTC_FAST area, which is included in the dump.

Example

1. In Project Configuration Menu, enable COREDUMP TO FLASH, then save and exit.
2. In your project, create a global variable in the DRAM area, such as:

```c
// uint8_t global_var;
COREDUMP_DRAM_ATTR uint8_t global_var;
```

3. In the main application, set the variable to any value and `assert(0)` to cause a crash.

```c
global_var = 25;
assert(0);
```

4. Build, flash, and run the application on a target device and wait for the dumping information.
5. Run the command below to start core dumping in GDB, where PORT is the device USB port:

```bash
idf.py coredump-debug
```

6. In GDB shell, type `p global_var` to get the variable content:

```gdb
(p) p global_var
$1 = 25 '\031'
```

4.7.8 Running `idf.py coredump-info` and `idf.py coredump-debug`

`idf.py coredump-info --help` and `idf.py coredump-debug --help` commands can be used to get more details on usage.

Related Documents

Anatomy of Core Dump Image

A core dump file’s format can be configured to use the ELF format, or a legacy binary format. The ELF format is recommended for all new designs as it provides more information regarding the software’s state at the moment the crash occurs, e.g., CPU registers and memory contents.

The memory state embeds a snapshot of all tasks mapped in the memory space of the program. The CPU state contains register values when the core dump has been generated. The core dump file uses a subset of the ELF structures to register this information.

Loadable ELF segments are used to store the process’ memory state, while ELF notes (`ELF.PT_NOTE`) are used to store the process’ metadata (e.g., PID, registers, signal etc). In particular, the CPU’s status is stored in a note with a special name and type (`CORE, NT_PRSTATUS type`).

Here is an overview of the core dump layout:

**Note:** The format of the image file shown in the above pictures represents the current version of the image and can be changed in future releases.
Chapter 4. API Guides

Fig. 3: Core Dump ELF Image Format

Fig. 4: Core Dump Binary Image Format
Overview of Implementation

The figure below describes some basic aspects related to the implementation of the core dump:

![Core Dump Implementation Overview](image)

**Note:** The diagram above hides some details and represents the current implementation of the core dump which can be changed later.

### 4.8 Current Consumption Measurement of Modules

You may want to know the current consumption of a module in deep-sleep mode, other power-saving modes, and active mode to develop some applications sensitive to power consumption. This section introduces how to measure the current consumption of a module running such an application.

#### 4.8.1 Notes to Measurement

**Can We Use a Development Board?**

For ESP32-S2, using a development board directly to measure current consumption of the corresponding module is not recommended, as some circuits still consume power on the board even when you flash the chip with the `deep_sleep` example. Therefore, you need to cut off the power supply circuit to the module to measure the module’s current. This method is inconvenient and increases measurement costs.
How to Choose an Appropriate Ammeter?

In the deep_sleep example, the module will be woken up every 20 seconds. In deep-sleep mode, the current in the module is just several microamps (μA), while in active mode, the current is in the order of milliamps (mA). The high dynamic current range makes accurate measurement difficult. Ordinary ammeters cannot dynamically switch the measurement range fast enough.

Additionally, ordinary ammeters have a relatively high internal resistance, resulting in a significant voltage drop. This may cause the module to enter an unstable state, as it is powered by a voltage smaller than the minimum required voltage supply.

Therefore, an ammeter suitable for measuring current in deep-sleep mode should have low internal resistance and, ideally, switch current ranges dynamically. We recommend two options: the Joulescope ammeter and the Power Profiler Kit II from Nordic.

Joulescope Ammeter  The Joulescope ammeter combines high-speed sampling and rapid dynamic current range switching to provide accurate and seamless current and energy measurements, even for devices with rapidly varying current consumption. Joulescope accurately measures electrical current over nine orders of magnitude from amps down to nanoamps. This wide range allows for accurate and precise current measurements for devices. Additionally, Joulescope has a total voltage drop of 25 mV at 1 A, which keeps the module running normally. These two features make Joulescope a perfect option for measuring the module switching between deep-sleep mode and wake-up mode.

Joulescope has no display screen. You need to connect it to a PC to visualize the current waveforms of the measured module. For specific instructions, please follow the documentation provided by the manufacturer.

Nordic Power Profiler Kit II  The Nordic Power Profiler Kit II has an advanced analog measurement unit with a high dynamic measurement range. This allows for accurate power consumption measurements for the entire range typically seen in low-power embedded applications, all the way from single μAs to 1 A. The resolution varies between 100 nA and 1 mA, depending on the measurement range, and is high enough to detect small spikes often seen in low-power optimized systems.

4.8.2 Hardware Connection

To measure the power consumption of a bare module, you need an ESP-Prog to flash the deep_sleep example to the module and power the module during measurement, a suitable ammeter (here we use the Joulescope ammeter), a computer, and of course a bare module with necessary jumper wires. For the connection, please refer to the following figure.

Please connect the pins of UART TX, UART RX, SPI Boot, Enable, and Ground on the measured module with corresponding pins on ESP-Prog, and connect the VPROG pin on ESP-Prog with the IN+ port on the Joulescope ammeter and connect its OUT+ port with the Power supply (3V3) pin on the measured module. For the specific names of these pins in different modules, please refer to the list below.

<table>
<thead>
<tr>
<th>Function of Module Pin</th>
<th>Pin Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART TX</td>
<td>TXD0</td>
</tr>
<tr>
<td>UART RX</td>
<td>RXD0</td>
</tr>
<tr>
<td>SPI Boot</td>
<td>IO0</td>
</tr>
<tr>
<td>Enable</td>
<td>EN</td>
</tr>
<tr>
<td>Power Supply</td>
<td>3V3</td>
</tr>
<tr>
<td>Ground</td>
<td>GND</td>
</tr>
</tbody>
</table>

For details of the pin names, please refer to the datasheet of specific module.
Fig. 6: Hardware Connection (click to enlarge)
4.8.3 Measurement Steps

ESP32-S3-WROOM-1 is used as an example in the measurement, and other modules can be measured similarly. For the specific current consumption of chips in different modes, please refer to the Current Consumption subsection in the corresponding chip datasheet.

You can refer to the following steps to measure the current in deep-sleep mode.

- Connect the aforementioned devices according to the hardware connection.
- Flash the `deep_sleep` example to the module. For details, please refer to Start a Project on Linux and macOS for a computer with Linux or macOS system or Start a Project on Windows for a computer with Windows system.

Please note that when you configure the example by running `idf.py menuconfig`, you need to disable `Enable touch wake up` in the Example Configuration to reduce the bottom current.

- By default, the module will be woken up every 20 seconds (you can change the timing by modifying the code of this example). To check if the example runs as expected, you can monitor the module operation by running `idf.py -p PORT monitor` (please replace `PORT` with your serial port name).
- Open the Joulescope software to see the current waveform as shown in the image below.

From the waveforms, you can obtain that the current of the module in deep-sleep mode is 8.14 μA. In addition, you can also see the current of the module in active mode, which is about 23.88 mA. The waveforms also show that the average power consumption during deep-sleep mode is 26.85 μW, and the average power consumption during active mode is 78.32 mW.

The figure below shows the total power consumption of one cycle is 6.37 mW.

By referring to these power consumption in different modes, you can estimate the power consumption of your applications and choose the appropriate power source.

4.9 Deep Sleep Wake Stubs

ESP32-S2 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.

4.9.1 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.

4.9.2 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
Fig. 7: Current Waveform of ESP32-S3-WROOM-1 (click to enlarge)
Fig. 8: Power Consumption of ESP32-S3-WROOM-1 (click to enlarge)
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_sleep.h` header under components/esp32s2.

### 4.9.3 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:

```c
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.

### 4.9.4 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:

```c
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
    esp_rom_printf(fmt_str, wake_count++);
}
```

The RTC memory area where this data will be placed can be configured via menuconfig option named `CONFIG_ESP32S2_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 attributes `RTC_FASTATTR` and `RTC_SLOWATTR` 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_FASTATTR` 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 \texttt{rtc\_wake\_stub\_counter.c}:

```c
int wake_count;

void RTC\_IRAM\_ATTR esp\_wake\_deep\_sleep(void) {
    esp\_default\_wake\_deep\_sleep();
    esp\_rom\_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 \texttt{CONFIG\_BOOTLOADER\_SKIP\_VALIDATE\_IN\_DEEP\_SLEEP} Kconfig option, see more information in \textit{Fast boot from Deep Sleep}.

### 4.9.5 CRC Check For Wake Stubs

During deep sleep, all RTC Fast memory areas will be validated with CRC. When ESP32-S2 wakes up from deep sleep, the RTC fast memory will be validated with CRC again. If the validation passes, the wake stubs code will be executed. Otherwise, the normal initialization, bootloader and esp-idf codes will be executed.

\textbf{Note:} When the \texttt{CONFIG\_ESP\_SYSTEM\_ALLOW\_RTC\_FAST\_MEM\_AS\_HEAP} option is enabled, all the RTC fast memory except the wake stubs area is added to the heap.

### 4.9.6 Example

ESP-IDF provides an example to show how to implement the Deep-sleep wake stub.

- \texttt{system/deep\_sleep\_wake\_stub}

### 4.10 Device Firmware Upgrade via USB

Typically, the firmware of the ESP32-S2 is flashed via the chip’s serial port. However, flashing via the serial port requires a USB to serial converter chip (e.g., CP210x or FTDI) to be connected to the ESP32-S2 (see \textit{Establish Serial Connection with ESP32-S2} for more details). The ESP32-S2 contains a USB OTG peripheral making it possible to connect the ESP32-S2 to the host directly via USB (thus not requiring a USB to serial converter chip).

Device Firmware Upgrade (DFU) is a mechanism for upgrading the firmware of the ESP32-S2 directly via the Universal Serial Bus (USB). However, enabling Secure Boot or flash encryption disables the USB-OTG USB stack in the ROM, disallowing updates via the serial emulation or DFU on that port.

- \textit{Software} of the Getting Started Guide introduces the software requirements of DFU.
- Section \textit{Building the DFU Image} describes how to build firmware for DFU with ESP-IDF.
- Section \textit{Flashing the DFU Image} deals with flashing the firmware.

### 4.10.1 USB Connection

The necessary connections for the ESP32-S2’s internal USB PHY (transceiver) are shown in the following table:

<table>
<thead>
<tr>
<th>GPIO</th>
<th>USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>D+ (green)</td>
</tr>
<tr>
<td>19</td>
<td>D- (white)</td>
</tr>
<tr>
<td>GND</td>
<td>GND (black)</td>
</tr>
<tr>
<td>+5V</td>
<td>+5V (red)</td>
</tr>
</tbody>
</table>
Warning: Some cables are wired up with non-standard colors and some drivers are able to work with swapped D+ and D- connections. Please try to swap the cables connecting to D+ and D- if your device is not detected.

Note: The ESP32-S2 chip needs to be in bootloader mode before it can be detected as a DFU device and flash. This can be achieved by pulling GPIO0 down (e.g., pressing the BOOT button), pulling RESET down for a moment, and releasing GPIO0.

4.10.2 Building the DFU Image

The command below will create a DFU image named dfu.bin that is placed in the project’s build directory:

```
idf.py dfu
```

Note: Do not forget to set the target chip by `idf.py set-target` before running `idf.py dfu`. Otherwise, you might create an image for a different chip or receive an error message like `unknown target 'dfu'`.

4.10.3 Flashing the DFU Image

The command below will download the DFU image into the ESP32-S2:

```
idf.py dfu-flash
```

The command relies on dfu-util. Please see Software for installing dfu-util. dfu-util needs additional setup for USB Drivers (Windows Only) or setting up an Udev Rule (Linux Only). macOS users should be able to use dfu-util without further setup.

If there are more boards with the same chip connected then `idf.py dfu-list` can be used to list the available devices, for example:

```
Found Runtime: [303a:0002] ver=0723, devnum=4, cfg=1, intf=2, path="1-10", alt=0,
               → name="UNKNOWN", serial="0"
Found Runtime: [303a:0002] ver=0723, devnum=6, cfg=1, intf=2, path="1-2", alt=0,
               → name="UNKNOWN", serial="0"
```

Consequently, the desired device can be selected for flashing by the `--path` argument. For example, the devices listed above can be flashed individually by the following commands:

```
idf.py dfu-flash --path 1-10
idf.py dfu-flash --path 1-2
```

Note: The vendor and product identifiers are set based on the selected chip target by the `idf.py set-target` command and they are not selectable during the `idf.py dfu-flash` call.

See Common Errors and Known Issues and their solutions.

4.10.4 Udev Rule (Linux Only)

Udev is a device manager for the Linux kernel. It allows us to run dfu-util (and idf.py dfu-flash) without sudo for gaining access to the chip.

Create file `/etc/udev/rules.d/40-dfuse.rules` with the following content:
SUBSYSTEMS="usb", ATTRS{idVendor}="303a", ATTRS{idProduct}="00??", GROUP="plugdev", MODE="0666"

**Note:** Please check the output of the command `groups`. The user has to be a member of the GROUP specified above. You may use some other existing groups for this purpose (e.g., `uucp` on some systems instead of `plugdev`) or create a new group for this purpose.

Restart your computer so the previous setting could take into affect or run `sudo udevadm trigger` to force manually udev to trigger your new rule.

### 4.10.5 USB Drivers (Windows Only)

`dfu-util` uses `libusb` to access the device. You have to register on Windows the device with the **WinUSB** driver. Please see the [libusb wiki](https://www.libusb.org) for more details.

The drivers can be installed by the **Zadig tool**. Please make sure that the device is in download mode before you run the tool and that it detects the ESP32-S2 device before you install the drivers. The Zadig tool might detect several USB interfaces of ESP32-S2. Please install the WinUSB driver only for the interface where there is no driver installed (probably it is Interface 2) and do not re-install the driver for the other interface.

**Warning:** The manual installation of the driver in Device Manager of Windows is not recommended because the flashing might not work properly.

### 4.10.6 Common Errors and Known Issues

- `dfu-util: command not found` might indicate that the tool has not been installed or is not available from the terminal. An easy way of checking the tool is running `dfu-util --version`. Please see [Software](https://www.espressif.com) for installing `dfu-util`.
- The reason for **No DFU capable USB device available** could be that the USB driver was not properly installed on Windows (see **USB Drivers (Windows Only)**), udev rule was not setup on Linux (see **Udev Rule (Linux Only)**) or the device is not in bootloader mode.
- Flashing with `dfu-util` on Windows fails on the first attempt with error `Lost device after RESET?`. Please retry the flashing and it should succeed the next time.

### 4.10.7 Secure Download Mode

When Secure Download Mode is enabled, DFU is no longer possible. Please see **Flash Encryption** guide for more details.

### 4.11 Error Handling

#### 4.11.1 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)
Chapter 4. API Guides

- C++ exceptions, thrown using throw keyword
- Unrecoverable (fatal) errors:
  - Failed assertions (using assert macro and equivalent methods, see Assertions) 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.

4.11.2 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.

4.11.3 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.

4.11.4 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 ESP-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.

### 4.11.5 ESP_ERROR_CHECKWITHOUT_ABORT Macro

ESP_ERROR_CHECKWITHOUT_ABORT macro serves similar purpose as ESP_ERROR_CHECK, except that it will not call abort().

### 4.11.6 ESP_RETURN_ON_ERROR Macro

ESP_RETURN_ON_ERROR macro checks the error code, if the error code is not equal ESP_OK, it prints the message and returns.

### 4.11.7 ESP_GOTO_ON_ERROR Macro

ESP_GOTO_ON_ERROR macro checks the error code, if the error code is not equal ESP_OK, it prints the message, sets the local variable ret to the code, and then exits by jumping to goto_tag.

### 4.11.8 ESP_RETURN_ON_FALSE Macro

ESP_RETURN_ON_FALSE macro checks the condition, if the condition is not equal true, it prints the message and returns with the supplied err_code.

### 4.11.9 ESP_GOTO_ON_FALSE Macro

ESP_GOTO_ON_FALSE macro checks the condition, if the condition is not equal true, it prints the message, sets the local variable ret to the supplied err_code, and then exits by jumping to goto_tag.

### 4.11.10 CHECK MACROS Examples

Some examples:

```c
static const char* TAG = "Test";

esp_err_t test_func(void)
{
    esp_err_t ret = ESP_OK;

    ESP_ERROR_CHECK(x); // err message
    // printed if 'x' is not 'ESP_OK', and then 'abort()'.
    ESP_ERROR_CHECKWITHOUT_ABORT(x); // err message
    // printed if 'x' is not 'ESP_OK', without 'abort()'.
    ESP_RETURN_ON_ERROR(x, TAG, "fail reason 1"); // err message
    // printed if 'x' is not 'ESP_OK', and then function returns with code 'x'.
```

(continues on next page)
### Chapter 4. API Guides

(continued from previous page)

```c
ESP_GOTO_ON_ERROR(x, err, TAG, "fail reason 2");  // err message
// printed if 'x' is not 'ESP_OK', 'ret' is set to 'x', and then jumps to 'err'.
ESP_RETURN_ON_FALSE(a, err_code, TAG, "fail reason 3");  // err message
// printed if 'a' is not 'true', and then function returns with code 'err_code'.
ESP_GOTO_ON_FALSE(a, err_code, err, TAG, "fail reason 4");  // err message
// printed if 'a' is not 'true', 'ret' is set to 'err_code', and then jumps to...
// err.

err:
// clean up
    return ret;
```

**Note:** If the option `CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT` in Kconfig is enabled, the err message will be discarded, while the other action works as is.

The `ESP_RETURN_XX` and `ESP_GOTO_xx` macros cannot be called from ISR. While there are `xx_ISR` versions for each of them, e.g., `ESP_RETURN_ON_ERROR_ISR`, these macros could be used in ISR.

---

#### 4.11.11 Error Handling Patterns

1. **Attempt to recover.** Depending on the situation, we may try the following methods:
   - retry the call after some time;
   - attempt to de-initialize the driver and re-initialize it again;
   - fix the error condition using an out-of-band mechanism (e.g., reset an external peripheral which is not responding).

   **Example:**

   ```c
   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:**

   ```c
   sdmmc_card_t *card = calloc(1, sizeof(sdmmc_card_t));
   if (card == NULL) {
       return ESP_ERR_NO_MEM;
   }
   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 behavior 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));
```

### 4.11.12 C++ Exceptions

See *Exception Handling*.

### 4.12 ESP-WIFI-MESH

This guide provides information regarding the ESP-WIFI-MESH protocol. Please see the *ESP-WIFI-MESH API Reference* for more information about API usage.

#### 4.12.1 Overview

ESP-WIFI-MESH is a networking protocol built atop the Wi-Fi protocol. ESP-WIFI-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-WIFI-MESH is self-organizing and self-healing meaning the network can be built and maintained autonomously.

The ESP-WIFI-MESH guide is split into the following sections:

1. *Introduction*
2. *ESP-WIFI-MESH Concepts*
3. *Building a Network*
4. *Managing a Network*
5. *Data Transmission*
6. *Channel Switching*
7. *Performance*
8. *Further Notes*

#### 4.12.2 Introduction

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.

ESP-WIFI-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-WIFI-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-WIFI-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.
Fig. 9: Traditional Wi-Fi Network Architecture

Fig. 10: ESP-WIFI-MESH Network Architecture
4.12.3 ESP-WIFI-MESH Concepts

Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Any device that is or can be part of an ESP-WIFI-MESH network</td>
</tr>
<tr>
<td>Root Node</td>
<td>The top node in the network</td>
</tr>
<tr>
<td>Child Node</td>
<td>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).</td>
</tr>
<tr>
<td>Parent Node</td>
<td>The converse notion of a child node</td>
</tr>
<tr>
<td>Descendant Node</td>
<td>Any node reachable by repeated proceeding from parent to child</td>
</tr>
<tr>
<td>Sibling Nodes</td>
<td>Nodes that share the same parent node</td>
</tr>
<tr>
<td>Connection</td>
<td>A traditional Wi-Fi association between an AP and a station. A node in ESP-WIFI-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.</td>
</tr>
<tr>
<td>Upstream Connection</td>
<td>The connection from an node to its parent node</td>
</tr>
<tr>
<td>Downstream Connection</td>
<td>The connection from an node to one of its child nodes</td>
</tr>
<tr>
<td>Wireless Hop</td>
<td>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.</td>
</tr>
<tr>
<td>Subnetwork</td>
<td>A subnetwork is subdivision of an ESP-WIFI-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-WIFI-MESH network.</td>
</tr>
<tr>
<td>MAC Address</td>
<td>Media Access Control Address used to uniquely identify each node or router within an ESP-WIFI-MESH network.</td>
</tr>
<tr>
<td>DS</td>
<td>Distribution System (External IP Network)</td>
</tr>
</tbody>
</table>

Tree Topology

ESP-WIFI-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-WIFI-MESH allows nodes to simultaneously act as a station and an AP. Therefore a node in ESP-WIFI-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-WIFI-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-WIFI-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-WIFI-MESH network can communicate.

Note: The size (total number of nodes) in an ESP-WIFI-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.

Node Types

Root Node: The root node is the top node in the network and serves as the only interface between the ESP-WIFI-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-WIFI-MESH network. There can only be one root node.
Chapter 4. API Guides

Fig. 11: ESP-WIFI-MESH Tree Topology

Maximum Permitted Layers = 4

Fig. 12: ESP-WIFI-MESH Node Types
**root node within an ESP-WIFI-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 network’s 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-WIFI-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-WIFI-MESH uses the Vendor Information Element to store metadata such as:

- Node Type (Root, Intermediate Parent, Leaf, Idle)
- 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-WIFI-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-WIFI-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-WIFI-MESH network 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.

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-WIFI-MESH network hence is always the preferred parent node 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-WIFI-MESH network will maintain its individual routing table used to correctly route ESP-WIFI-MESH packets (see ESP-WIFI-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.

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-WIFI-MESH utilizes routing tables to determine whether an ESP-WIFI-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.

4.12.4 Building a Network

General Process

**Warning:** Before the ESP-WIFI-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-WIFI-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-WIFI-MESH network building process can be generalized into the following steps:

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 number of 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-WIFI-MESH network is built when the root node is automatically selected.

![Diagram of ESP-WIFI-MESH network with root node selection process](image-url)
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 (-10 dB) 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-WIFI-MESH network leading to the building of multiple networks. If such is the case, ESP-WIFI-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-WIFI-MESH network is built when the root node is designated by the user.

![Diagram of root node designation](image)

Fig. 18: 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-WIFI-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-WIFI-MESH network and to balance the number of downstream connections between potential parent nodes (see section on Preferred Parent Node).

However ESP-WIFI-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-WIFI-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 (-20 dB and -10 dB) 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.
Fig. 19: Network Building with Asynchronous Power On Example
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 (-10 dB). 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()`).

**Note:** Differences in parent node selection caused by asynchronous power-on are autonomously corrected for to some extent in ESP-WIFI-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-WIFI-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-WIFI-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.

---

**4.12.5 Managing a Network**

**ESP-WIFI-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-WIFI-MESH will autonomously select a new parent node and form an upstream connection with it to maintain network interconnectivity. ESP-WIFI-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.

1. Node C is the root node of the network. Nodes A/B/D/E are second layer nodes connected to node C.
2. 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.

**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-WIFI-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. 22: 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 network’s 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.
Chapter 4. API Guides

Note: Root node switching must require an election hence is only supported when using a self-organized ESP-WIFI-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-WIFI-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.

4.12.6 Data Transmission

ESP-WIFI-MESH Packet

ESP-WIFI-MESH network data transmissions use ESP-WIFI-MESH packets. ESP-WIFI-MESH packets are entirely contained within the frame body of a Wi-Fi data frame. A multi-hop data transmission in an ESP-WIFI-MESH network will involve a single ESP-WIFI-MESH packet being carried over each wireless hop by a different Wi-Fi data frame.

The following diagram shows the structure of an ESP-WIFI-MESH packet and its relation with a Wi-Fi data frame.

![ESP-WIFI-MESH Packet Diagram](image)

**Fig. 23: ESP-WIFI-MESH Packet**

The **header** of an ESP-WIFI-MESH packet contains the MAC addresses of the source and destination nodes. The options field contains information pertaining to the special types of ESP-WIFI-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-WIFI-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-WIFI-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-WIFI-MESH packet to be transmitted simultaneously to multiple nodes within the network. Multicasting in ESP-WIFI-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-WIFI-MESH packet’s destination address to the Multicast-Group Address (01:00:5E:xx:xx:xx). This signifies that the ESP-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-MESH Bidirectional Data Stream.

![Fig. 24: ESP-WIFI-MESH Bidirectional Data Stream](image)

Due to the use of **Routing Tables**, ESP-WIFI-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.

**4.12.7 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-WIFI-MESH Network Channel Switching

ESP-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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-WIFI-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 their own beacon frames.

#### Router Triggered

When an ESP-WIFI-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-WIFI-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-WIFI-MESH network. By using a custom value, nodes within the ESP-WIFI-MESH network are able to switch channels synchronously relative to the ESP-WIFI-MESH network’s beacon interval. However, this will also result in the ESP-WIFI-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-WIFI-MESH network channel switch being unsynchronized with the router’s channel switch, there will be:
  - The ESP-WIFI-MESH network’s channel switch time is dependent on the ESP-WIFI-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-WIFI-MESH network’s switch.

- In the ESP-WIFI-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-WIFI-MESH network’s performance. Note that this can be caused by the ESP-WIFI-MESH network itself (e.g., due to wireless medium contention with ESP-WIFI-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), the router will switch channels without the ESP-WIFI-MESH network’s knowledge.**

  - 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-WIFI-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-WIFI-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-WIFI-MESH network to dynamically switch channels when set.

**Preset Channel:** An ESP-WIFI-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-WIFI-MESH to dynamically switch to a different router when set.

**Preset Router BSSID:** An ESP-WIFI-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 “do not care” for the parameter.
### 4.12.8 Performance

The performance of an ESP-WIFI-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-WIFI-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-WIFI-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-WIFI-MESH network:

<table>
<thead>
<tr>
<th>Preset Channel Switch</th>
<th>Allow Channel Switch</th>
<th>Preset Router BSSID</th>
<th>Allow Router Switch</th>
<th>Root Node Present</th>
<th>Permitted Switches?</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>X</td>
<td>N</td>
<td>X</td>
<td>X</td>
<td>Channel and Router</td>
</tr>
<tr>
<td>N</td>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>X</td>
<td>Channel Only</td>
</tr>
<tr>
<td>N</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td>X</td>
<td>Channel and Router</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>X</td>
<td>X</td>
<td>Channel and Router</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>X</td>
<td>N</td>
<td>Router Only</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>X</td>
<td>Y</td>
<td>Channel and Router</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>X</td>
<td>Channel Only</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Network Only</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Channel Only</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Router Only</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Channel and Router</td>
</tr>
</tbody>
</table>

**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-WIFI-MESH network and the bandwidth of the router.
**Note:** The performance figures can vary greatly between installations based on network configuration and operating environment.

### 4.12.9 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

### 4.13 Support for External RAM

#### 4.13.1 Introduction

ESP32-S2 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-S2 has the ability to use up to 10.5 MB of virtual addresses for external PSRAM (Pseudo static 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.

#### 4.13.2 Hardware

ESP32-S2 supports PSRAM connected in parallel with the SPI flash chip. While ESP32-S2 is capable of supporting several types of RAM chips, ESP-IDF currently only supports Espressif branded PSRAM chips (e.g., ESP-PSRAM32, ESP-PSRAM64, etc).

**Note:** Some PSRAM chips are 1.8 V devices and some are 3.3 V. The working voltage of the PSRAM chip must match the working voltage of the flash component. Consult the datasheet for your PSRAM chip and ESP32-S2 device to find out the working voltages. For a 1.8 V PSRAM chip, make sure to either set the MTDI pin to a high signal level on bootup, or program ESP32-S2 eFuses to always use the VDD_SIO level of 1.8 V. Not doing this can damage the PSRAM and/or flash chip.

**Note:** Espressif produces both modules and system-in-package chips that integrate compatible PSRAM and flash and are ready to mount on a product PCB. Consult the Espressif website for more information. If you are using a custom PSRAM chip, ESP-IDF SDK might not be compatible with it.

For specific details about connecting the SoC or module pins to an external PSRAM chip, consult the SoC or module datasheet.

#### 4.13.3 Configuring External RAM

ESP-IDF fully supports the use of external RAM in applications. Once the external RAM is initialized at startup, ESP-IDF can be configured to integrate the external RAM in several ways:

- *Integrate RAM into the ESP32-S2 Memory Map*
- *Add External RAM to the Capability Allocator*
- *Provide External RAM via malloc() (default)*
Integrate RAM into the ESP32-S2 Memory Map

Select this option by choosing Integrate RAM into memory map from CONFIG_SPIRAM_USE.

This is the most basic option for external RAM integration. Most likely, you will need another, more advanced option. During the ESP-IDF startup, external RAM is mapped into the data virtual address space. The address space is dynamically allocated. The length will be the minimum length between the PSRAM size and the available data virtual address space size.

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 RAM: coordinating buffer usage, preventing corruption, etc.

It is recommended to access the PSRAM by ESP-IDF heap memory allocator (see next chapter).

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 data virtual address space 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 to Be Placed in External Memory

Enable this option by checking CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY.

If enabled, the region of the data virtual address space where the PSRAM is mapped to will be used to store zero-initialized data (BSS segment) from the lwIP, net80211, libpp, and bluedroid ESP-IDF libraries.
Additional data can be moved from the internal BSS segment to external RAM by applying the macro `EXT_RAM_BSS_ATTR` to any static declaration (which is not initialized to a non-zero value).

It is also possible to place the BSS section of a component or a library to external RAM using linker fragment scheme `extram_bss`.

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.

**Move Instructions in Flash to PSRAM**

The `CONFIG_SPIRAM_FETCH_INSTRUCTIONS` option allows the flash `.text` sections (use for instructions) to be placed in PSRAM.

By enabling the `CONFIG_SPIRAM_FETCH_INSTRUCTIONS` option

- Instructions from the `.text` sections of flash are moved into PSRAM on system startup.
- The corresponding virtual memory range of those instructions will also be re-mapped to PSRAM.

If `CONFIG_SPIRAM_RODATA` is also enabled, the cache will not be disabled during an SPI flash operation. You do not need to make sure ISRs, ISR callbacks and involved data are placed in internal RAM, thus internal RAM usage can be optimized.

**Move Read-Only Data in Flash to PSRAM**

The `CONFIG_SPIRAM_RODATA` option allows the flash `.rodata` sections (use for read only data) to be placed in PSRAM.

By enabling the `CONFIG_SPIRAM_RODATA` option

- Instructions from the `.rodata` sections of flash are moved into PSRAM on system startup.
- The corresponding virtual memory range of those rodata will also be re-mapped to PSRAM.

If `CONFIG_SPIRAM_FETCH_INSTRUCTIONS` is also enabled, the cache will not be disabled during an SPI flash operation. You do not need to make sure ISRs, ISR callbacks and involved data are placed in internal RAM, thus internal RAM usage can be optimized.

**4.13.4 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 read operations from or write operations 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. Therefore when External RAM is enabled, any buffer that will be used in combination with DMA must be allocated using `heap_caps_malloc(size, MALLOC_CAP_DMA | MALLOC_CAP_INTERNAL)` and can be freed using a standard `free()` call. Note that although ESP32-S2 has hardware support for DMA to or from external RAM, this is not yet supported in ESP-IDF.
- 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.
- In general, external RAM will not be used as task stack memory. `xTaskCreate()` and similar functions will always allocate internal memory for stack and task TCBs.
The option `CONFIG_SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY` can be used to allow placing task stacks into external memory. In these cases `xTaskCreateStatic()` must be used to specify a task stack buffer allocated from external memory, otherwise task stacks will still be allocated from internal memory.

4.13.5 Failure to Initialize

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.

4.13.6 Encryption

It is possible to enable automatic encryption for data stored in external RAM. When this is enabled any data read and written through the cache will automatically be encrypted or decrypted by the external memory encryption hardware.

This feature is enabled whenever flash encryption is enabled. For more information on how to enable and how it works see Flash Encryption.

4.14 Fatal Errors

4.14.1 Overview

In certain situations, the execution of the program cannot 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
  - Memory protection fault
  - Brownout detection event
  - Stack overflow
  - Stack smashing protection check
  - Heap integrity check
  - Undefined behavior sanitizer (UBSAN) checks
- 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.

4.14.2 Panic Handler

Every error cause listed in the Overview will be handled by the panic handler.

The 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). Exception was unhandled.

In all cases, the error cause will be printed in parentheses. See *Guru Meditation Errors* for a list of possible error causes.

Subsequent behavior of the panic handler can be set using `CONFIG_ESP_SYSTEM_PANIC` configuration choice. The available options are:

- **Print registers and reboot** (`CONFIG_ESP_SYSTEM_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_ESP_SYSTEM_PANIC_PRINT_HALT`) Similar to the above option, but halt instead of rebooting. External reset is required to restart the program.
- **Silent reboot** (`CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT`) Do not print registers or backtrace, restart the chip immediately.
- **Invoke GDB Stub** (`CONFIG_ESP_SYSTEM_PANIC_GDBSTUB`) Start GDB server which can communicate with GDB over console UART port. This option will only provide read-only debugging or post-mortem debugging. See *GDB Stub* for more details.

The behavior of the panic handler is affected by three other configuration options.

- If `CONFIG_ESP_DEBUG_OCD_AWARE` is enabled (which is the default), the 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 the **Core Dump** feature is enabled, then the system state (task stacks and registers) will be dumped to either 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 (including required UART functions) is placed in IRAM, and hence will decrease the usable memory space in SRAM. But this may be necessary to debug some complex issues with crashes while flash cache is disabled (for example, when writing to SPI flash) or when flash cache is corrupted when an exception is triggered.
- If `CONFIG_ESP_SYSTEM_PANIC_REBOOT_DELAY_SECONDS` is enabled (disabled by default) and set to a number higher than 0, the panic handler will delay the reboot for that amount of time in seconds. This can help if the tool used to monitor serial output does not provide a possibility to stop and examine the serial output. In that case, delaying the reboot will allow users to examine and debug the panic handler output (backtrace, etc.) for the duration of the delay. After the delay, the device will reboot. The reset reason is preserved.

The following diagram illustrates the panic handler behavior:

### 4.14.3 Register Dump and Backtrace

Unless the `CONFIG_ESP_SYSTEM_PANIC_SILENT_REBOOT` option is enabled, the panic handler prints some of the CPU registers, and the backtrace, to the console

---

(continues on next page)
Fig. 25: Panic Handler Flowchart (click to enlarge)
The register values printed are the register values in the exception frame, i.e., values at the moment when the CPU exception or another fatal error has occurred.

A Register dump is not printed if the panic handler has been executed as a result of an abort() call.

In some cases, such as interrupt watchdog timeout, the panic handler may print additional CPU registers (EPC1-EPC4) and the registers/backtrace of the code running on the other CPU.

The 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.

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:

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.

### 4.14.4 GDB Stub

If the CONFIG_ESP_SYSTEM_PANIC_GDBSTUB option is enabled, the panic handler will not reset the chip when a fatal error happens. Instead, it will start a GDB remote protocol server, commonly referred to as GDB Stub. When this happens, a GDB instance running on the host computer can be instructed to connect to the ESP32-S2 UART port.

If IDF Monitor is used, GDB is started automatically when a GDB Stub prompt is detected on the UART. The output looks like this:

Entering gdb stub now.
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-
esp32s2-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
Find the GDB manual and other documentation resources online at:
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.
0x400e1b41 in app_main ()
at /Users/user/esp/example/main/main.cpp:36
36    *((int*) 0) = 0;
(gdb)

The 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 the PC, or continue execution. To reset the program, exit GDB and perform an external reset: Ctrl-T Ctrl-R in IDF Monitor, or using the external reset button on the development board.

### 4.14.5 RTC Watchdog Timeout

The RTC watchdog is used in the startup code to keep track of execution time and it also helps to prevent a lock-up caused by an unstable power source. It is enabled by default (see `CONFIG_BOOTLOADER_WDT_ENABLE`). If the execution time is exceeded, the RTC watchdog will restart the system. In this case, the ROM bootloader will print a message with the RTC Watchdog Timeout reason for the reboot.

```
rst:0x10 (RTCWDT_RTC_RST)
```

The RTC watchdog covers the execution time from the first stage bootloader (ROM bootloader) to application startup. It is initially set in the ROM bootloader, then configured in the bootloader with the `CONFIG_BOOTLOADER_WDT_TIME_MS` option (9000 ms by default). During the application initialization stage, it is reconfigured because the source of the slow clock may have changed, and finally disabled right before the `app_main()` call. There is an option `CONFIG_BOOTLOADER_WDT_DISABLE_IN_USER_CODE` which prevents the RTC watchdog from being disabled before `app_main`. Instead, the RTC watchdog remains active and must be fed periodically in your application’s code.

### 4.14.6 Guru Meditation Errors

This section explains the meaning of different error causes, printed in parens after the Guru Meditation Error: Core panic’ed message.

**Note:** See the [Guru Meditation Wikipedia article](http://www.guru-meditation.org) for historical origins of “Guru Meditation”.

**IllegalInstruction**

This CPU exception indicates that the instruction which was executed was not a valid instruction. The most common reasons for this error include:

- FreeRTOS task function has returned. In FreeRTOS, if a task function needs to terminate, it should call `vTaskDelete()` and delete itself, instead of returning.
- Failure to read next instruction from SPI flash. This usually happens if:
- Application has reconfigured the SPI flash pins as some other function (GPIO, UART, etc.). Consult the Hardware Design Guidelines and the datasheet for the chip or module for details about the SPI flash pins.
- Some external device has accidentally been connected to the SPI flash pins, and has interfered with communication between ESP32-S2 and SPI flash.

- In C++ code, exiting from a non-void function without returning a value is considered to be an undefined behavior. When optimizations are enabled, the compiler will often omit the epilogue in such functions. This most often results in an IllegalInstruction exception. By default, ESP-IDF build system enables `-Werror=return-type` which means that missing return statements are treated as compile time errors. However if the application project disables compiler warnings, this issue might go undetected and the IllegalInstruction exception will occur at run time.

**InstrFetchProhibited**

This CPU exception indicates that the CPU could not read an instruction because the address of the instruction does 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 a garbage value (not 0x4xxxxxxxx).

**LoadProhibited, StoreProhibited**

These CPU exceptions happen when an application attempts to read from or write to an invalid memory location. The address which has been written/read is found in the EXCVADDR register in the register dump. If this address is zero, it usually means that the application has attempted to dereference a NULL pointer. If this address is close to zero, it usually means that the application has attempted to access a member of a structure, but the pointer to the structure is NULL. If this address is something else (garbage value, not in 0x3fxxxxxx - 0x6xxxxxxx range), it likely means that the pointer used to access the data is either not initialized or has been corrupted.

**IntegerDivideByZero**

Application has attempted to do an integer division by zero.

**LoadStoreAlignment**

Application has attempted to read or write a memory location, and the address alignment does not match the load/store size. For example, a 32-bit read can only be done from a 4-byte aligned address, and a 16-bit write can only be done to a 2-byte aligned address.

**LoadStoreError**

This exception may happen in the following cases:

- If the application has attempted to do an 8- or 16-bit read to, or write from, a memory region which only supports 32-bit reads/writes. For example, dereferencing a char* pointer to instruction memory (IRAM, IROM) will result in such an error.

- If the application has attempted to write to a read-only memory region, such as IROM or DROM.

**Unhandled Debug Exception**

This CPU exception happens when the instruction `BREAK` is executed.

**Interrupt Watchdog Timeout on CPU0/CPU1**

Indicates that an 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 when 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.

Memory Protection Fault

ESP32-S2 Permission Control feature is used in ESP-IDF to prevent the following types of memory access:

• writing to instruction RAM after the program is loaded
• executing code from data RAM (areas used for heap and static .data and .bss)

Such operations are not necessary for most programs. Prohibiting such operations typically makes software vulnerabilities harder to exploit. Applications which rely on dynamic loading or self-modifying code may disable this protection using CONFIG_ESP_SYSTEM_MEMPROT_FEATURE Kconfig option.

When the fault occurs, the panic handler reports the address of the fault and the type of memory access that caused it.

4.14.7 Other Fatal Errors

Brownout

ESP32-S2 has a built-in brownout detector, which is enabled by default. The brownout detector can trigger a system reset if the supply voltage goes below a safe level. The brownout detector can be configured using CONFIG_ESP_BROWNOUT_DET and CONFIG_ESP_BROWNOUT_DET_LVL_SEL options.

When the brownout detector triggers, the following message is printed:

```
Brownout detector was triggered
```

The chip is reset after the message is printed.

Note that if the supply voltage is dropping at a fast rate, only part of the message may be seen on the console.

Corrupt Heap

ESP-IDF’s heap implementation contains a number of run-time checks of the heap structure. Additional checks (“Heap Poisoning”) can be enabled in menuconfig. If one of the checks fails, a 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 overflow

FreeRTOS End of Stack Watchpoint ESP-IDF provides a custom FreeRTOS stack overflow detecting mechanism based on watchpoints. Every time FreeRTOS switches task context, one of the watchpoints is set to watch the last 32 bytes of stack.
Generally, this may cause the watchpoint to be triggered up to 28 bytes earlier than expected. The value 32 is chosen because it is larger than the stack canary size in FreeRTOS (20 bytes). Adopting this approach ensures that the watchpoint triggers before the stack canary is corrupted, not after.

**Note:** Not every stack overflow is guaranteed to trigger the watchpoint. It is possible that the task writes to memory beyond the stack canary location, in which case the watchpoint will not be triggered.

If watchpoint triggers, the message will be similar to:

```
Debug exception reason: Stack canary watchpoint triggered (task_name)
```

This feature can be enabled by using the `CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK` option.

### FreeRTOS Stack Checks

See `CONFIG_FREERTOS_CHECK_STACKOVERFLOW`

#### 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
```

The backtrace should point to the function where stack smashing has occurred. Check the function code for unbounded access to local arrays.

#### Undefined Behavior Sanitizer (UBSAN) Checks

Undefined behavior sanitizer (UBSAN) is a compiler feature which adds run-time checks for potentially incorrect operations, such as:

- overflows (multiplication overflow, signed integer overflow)
- shift base or exponent errors (e.g., shift by more than 32 bits)
- integer conversion errors

See [GCC documentation](https://gcc.gnu.org/onlinedocs/gcc/Undefined-Behavior-Sanitizer.html) of `-fsanitize=undefined` option for the complete list of supported checks.

### Enabling UBSAN

UBSAN is disabled by default. It can be enabled at file, component, or project level by adding the `-fsanitize=undefined` compiler option in the build system.

When enabling UBSAN for code which uses the SOC hardware register header files (`soc/xxx_reg.h`), it is recommended to disable shift-base sanitizer using `-fno-sanitize=shift-base` option. This is due to the fact that ESP-IDF register header files currently contain patterns which cause false positives for this specific sanitizer option.

To enable UBSAN at project level, add the following code at the end of the project’s `CMakeLists.txt` file:

```
idf_build_set_property(COMPIL\E_OPTIONS "-fsanitize=undefined" "-fno-sanitize=shift-base" APPEND)
```
Alternatively, pass these options through the `EXTRA_CFLAGS` and `EXTRA_CXXFLAGS` environment variables.

Enabling UBSAN results in significant increase of code and data size. Most applications, except for the trivial ones, will not fit into the available RAM of the microcontroller when UBSAN is enabled for the whole application. Therefore it is recommended that UBSAN is instead enabled for specific components under test.

To enable UBSAN for a specific component (`component_name`) from the project’s `CMakeLists.txt` file, add the following code at the end of the file:

```cmake
idf_component_get_property(lib component_name COMPONENT_LIB)
target_compile_options(${lib} PRIVATE "-fsanitize=undefined" "-fno-sanitize=shift-base")
```

**Note:** See the build system documentation for more information about build properties and component properties.

To enable UBSAN for a specific component (`component_name`) from `CMakeLists.txt` of the same component, add the following at the end of the file:

```cmake
target_compile_options(${COMPONENT_LIB} PRIVATE "-fsanitize=undefined" "-fno-sanitize=shift-base")
```

**UBSAN Output** When UBSAN detects an error, a message and the backtrace are printed, for example:

```plaintext
Undefined behavior of type out_of_bounds
Backtrace:0x4008b383:panic_abort at /path/to/esp-idf/components/esp_system/panic.c:367
          0x4008c791:esp_system_abort at /path/to/esp-idf/components/esp_system/system_api.c:106
          0x4008c587:__ubsan_default_handler at /path/to/esp-idf/components/esp_system/ubsan.c:152
          0x4008c6be:__ubsan_handle_out_of_bounds at /path/to/esp-idf/components/esp_system/ubsan.c:223
          0x400db74f:test_ub at main.c:128
          0x400db99c:app_main at main.c:56 (discriminator 1)
```

The types of errors reported by UBSAN can be as follows:
### Name | Meaning
--- | ---
`type_mismatch`, `type_mismatch_v1` | Incorrect pointer value: null, unaligned, not compatible with the given type.
`add_overflow`, `sub_overflow`, `mul_overflow`, `negate_overflow` | Integer overflow during addition, subtraction, multiplication, negation.
`divrem_overflow` | Integer division by 0 or `INT_MIN`.
`shift_out_of_bounds` | Overflow in left or right shift operators.
`out_of_bounds` | Access outside of bounds of an array.
`unreachable` | Unreachable code executed.
`missing_return` | Non-void function has reached its end without returning a value (C++ only).
`vla_bound_not_positive` | Size of variable length array is not positive.
`load_invalid_value` | Value of `bool` or `enum` (C++ only) variable is invalid (out of bounds).
`nonnull_arg` | Null argument passed to a function which is declared with a `nonnull` attribute.
`nonnull_return` | Null value returned from a function which is declared with `returnsnonnull` attribute.
`builtin_unreachable` | `__builtin_unreachable` function called.
`pointer_overflow` | Overflow in pointer arithmetic.

## 4.15 Hardware Abstraction

ESP-IDF provides a group of APIs for hardware abstraction. These APIs allow you to control peripherals at different levels of abstraction, giving you more flexibility compared to using only the ESP-IDF drivers to interact with hardware. ESP-IDF Hardware abstraction is likely to be useful for writing high-performance bare-metal drivers, or for attempting to port an ESP chip to another platform.

This guide is split into the following sections:

1. **Architecture**
2. **LL (Low Level) Layer**
3. **HAL (Hardware Abstraction Layer)**

**Warning:** Hardware abstraction API (excluding the driver and `xxx_types.h`) should be considered an experimental feature, thus cannot be considered public API. The hardware abstraction API does not adhere to the API name changing restrictions of ESP-IDF’s versioning scheme. In other words, it is possible that Hardware Abstraction API may change in between non-major release versions.

**Note:** Although this document mainly focuses on hardware abstraction of peripherals, e.g., UART, SPI, I2C, certain layers of hardware abstraction extend to other aspects of hardware as well, e.g., some of the CPU’s features are partially abstracted.

### 4.15.1 Architecture

Hardware abstraction in ESP-IDF is comprised of the following layers, ordered from low level of abstraction that is closer to hardware, to high level of abstraction that is further away from hardware.

- Low Level (LL) Layer
- Hardware Abstraction Layer (HAL)
- Driver Layers
The LL Layer, and HAL are entirely contained within the hal component. Each layer is dependent on the layer below it, i.e., driver depends on HAL, HAL depends on LL, LL depends on the register header files.

For a particular peripheral xxx, its hardware abstraction generally consists of the header files described in the table below. Files that are Target Specific have a separate implementation for each target, i.e., a separate copy for each chip. However, the #include directive is still target-independent, i.e., is the same for different targets, as the build system automatically includes the correct version of the header and source files.

<table>
<thead>
<tr>
<th>Include Directive</th>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include 'soc/xxx_caps.h&quot;</td>
<td>Y</td>
<td>This header contains a list of C macros specifying the various capabilities of the ESP32-S2’s peripheral xxx. Hardware capabilities of a peripheral include things such as the number of channels, DMA support, hardware FIFO/buffer lengths, etc.</td>
</tr>
<tr>
<td>#include &quot;soc/xxx_struct.h&quot;</td>
<td>Y</td>
<td>The two headers contain a representation of a peripheral’s registers in C structure and C macro format respectively, allowing you to operate a peripheral at the register level via either of these two header files.</td>
</tr>
<tr>
<td>#include &quot;soc/xxx_reg.h&quot;</td>
<td>Y</td>
<td>If certain signals of a peripheral are mapped to a particular pin of the ESP32-S2, their mappings are defined in this header as C macros.</td>
</tr>
<tr>
<td>#include &quot;soc/xxx_pins.h&quot;</td>
<td>Y</td>
<td>This header is mainly used as a convenience header file to automatically include xxx_caps.h, xxx_struct.h, and xxx_reg.h.</td>
</tr>
<tr>
<td>#include &quot;hal/xxx_periph.h&quot;</td>
<td>N</td>
<td>This header contains type definitions and macros that are shared among the LL, HAL, and driver layers. Moreover, it is considered public API thus can be included by the application level. The shared types and definitions usually related to non-implementation specific concepts such as the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Protocol-related types/macros such a frames, modes, common bus speeds, etc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Features/characteristics of an xxx peripheral that are likely to be present on any implementation (implementation-independent) such as channels, operating modes, signal amplification or attenuation intensities, etc.</td>
</tr>
<tr>
<td>#include &quot;hal/xxx_types.h&quot;</td>
<td>N</td>
<td>This header contains the Low Level (LL) Layer of hardware abstraction. LL Layer API are primarily used to abstract away register operations into readable functions.</td>
</tr>
<tr>
<td>#include &quot;hal/xxx_ll.h&quot;</td>
<td>Y</td>
<td>The Hardware Abstraction Layer (HAL) is used to abstract away peripheral operation steps into functions (e.g., reading a buffer, starting a transmission, handling an event, etc). The HAL is built on top of the LL Layer.</td>
</tr>
<tr>
<td>#include &quot;hal/xxx_hal.h&quot;</td>
<td>Y</td>
<td>The driver layer is the highest level of ESP-IDF’s hardware abstraction. Driver layer API are meant to be called from ESP-IDF applications, and internally utilize OS primitives. Thus, driver layer API are event-driven, and can used in a multi-threaded environment.</td>
</tr>
</tbody>
</table>

4.15.2 LL (Low Level) Layer

The primary purpose of the LL Layer is to abstract away register field access into more easily understandable functions. LL functions essentially translate various in/out arguments into the register fields of a peripheral in the form of get/set functions. All the necessary bit shifting, masking, offsetting, andendianness of the register fields should be handled by the LL functions.

```c
//Inside xxx_ll.h

static inline void xxx_ll_set_baud_rate(xxx_dev_t *hw,
```
Chapter 4. API Guides

The code snippet above illustrates typical LL functions for a peripheral xxx. LL functions typically have the following characteristics:

- All LL functions are defined as static inline so that there is minimal overhead when calling these functions due to compiler optimization. These functions are not guaranteed to be inlined by the compiler, so any LL function that is called when the cache is disabled (e.g., from an IRAM ISR context) should be marked with __attribute__((always_inline)).
- The first argument should be a pointer to a xxx_dev_t type. The xxx_dev_t type is a structure representing the peripheral’s registers, thus the first argument is always a pointer to the starting address of the peripheral’s registers. Note that in some cases where the peripheral has multiple channels with identical register layouts, xxx_dev_t *hw may point to the registers of a particular channel instead.
- LL functions should be short, and in most cases are deterministic. In other words, in the worst case, runtime of the LL function can be determined at compile time. Thus, any loops in LL functions should be finite bounded; however, there are currently a few exceptions to this rule.
- LL functions are not thread-safe, it is the responsibility of the upper layers (driver layer) to ensure that registers or register fields are not accessed concurrently.

4.15.3 HAL (Hardware Abstraction Layer)

The HAL layer models the operational process of a peripheral as a set of general steps, where each step has an associated function. For each step, the details of a peripheral’s register implementation (i.e., which registers need to be set/read) are hidden (abstracted away) by the HAL. By modeling peripheral operation as a set of functional steps, any minor hardware implementation differences of the peripheral between different targets or chip versions can be abstracted away by the HAL (i.e., handled transparently). In other words, the HAL API for a particular peripheral remains mostly the same across multiple targets/chip versions.

The following HAL function examples are selected from the Watchdog Timer HAL as each function maps to one of the steps in a WDT’s operation life cycle, thus illustrating how a HAL abstracts a peripheral’s operation into functional steps.

```c
// Initialize one of the WDTs
void wdt_hal_init(wdt_hal_context_t *hal, wdt_inst_t wdt_inst, uint32_t prescaler, bool enable_intr);

// Configure a particular timeout stage of the WDT
void wdt_hal_config_stage(wdt_hal_context_t *hal, wdt_stage_t stage, uint32_t timeout, wdt_stage_action_t behavior);

// Start the WDT
void wdt_hal_enable(wdt_hal_context_t *hal);

// Feed (i.e., reset) the WDT
void wdt_hal_feed(wdt_hal_context_t *hal);
```

(continues on next page)
HAL functions generally have the following characteristics:

- The first argument to a HAL function has the `xxx_hal_context_t *` type. The HAL context type is used to store information about a particular instance of the peripheral (i.e., the context instance). A HAL context is initialized by the `xxx_hal_init()` function and can store information such as the following:
  - The channel number of this instance
  - Pointer to the peripheral’s (or channel’s) registers (i.e., a `xxx_dev_t *` type)
  - Information about an ongoing transaction (e.g., pointer to DMA descriptor list in use)
  - Some configuration values for the instance (e.g., channel configurations)
  - Variables to maintain state information regarding the instance (e.g., a flag to indicate if the instance is waiting for transaction to complete)

- HAL functions should not contain any OS primitives such as queues, semaphores, mutexes, etc. All synchronization/concurrency should be handled at higher layers (e.g., the driver).
- Some peripherals may have steps that cannot be further abstracted by the HAL, thus end up being a direct wrapper (or macro) for an LL function.
- Some HAL functions may be placed in IRAM thus may carry an `IRAM_ATTR` or be placed in a separate `xxx_hal_iram.c` source file.

### 4.16 High Priority Interrupts

The Xtensa architecture supports 32 interrupts, divided over 7 priority levels from level 1 to 7, with level 7 being a non-maskable interrupt (NMI), plus an assortment of exceptions. On the ESP32-S2, the Interrupt Allocation can route most interrupt sources to these interrupts via the interrupt mux. Normally, interrupts are written in C, but ESP-IDF allows high-priority interrupts to be written in assembly as well, resulting in very low interrupt latencies.

#### 4.16.1 Interrupt Priorities

<table>
<thead>
<tr>
<th>Priority Level</th>
<th>Symbol</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N/A</td>
<td>Exception and low priority interrupts, handled by ESP-IDF.</td>
</tr>
<tr>
<td>2-3</td>
<td>N/A</td>
<td>Medium priority interrupts, handled by ESP-IDF.</td>
</tr>
<tr>
<td>4</td>
<td>xt_highint4</td>
<td>Normally used by ESP-IDF debug logic.</td>
</tr>
<tr>
<td>5</td>
<td>xt_highint5</td>
<td>High priority interrupts, free to use.</td>
</tr>
<tr>
<td>NMI</td>
<td>xt_nmi</td>
<td>Non-maskable interrupt, free to use.</td>
</tr>
<tr>
<td>dbg</td>
<td>xt_debugexception</td>
<td>Debug exception. Called on e.g., a BREAK instruction.</td>
</tr>
</tbody>
</table>

Using these symbols is done by creating an assembly file with suffix `.S` and defining the named symbols, like this:

```assembly
.section .iram1,"ax"
.global xt_highint5
.type xt_highint5, @function
.align 4
xt_highint5: ...
... your code here
```

(continues on next page)
For a real-life example, see the `esp_system/port/soc/esp32s2/highint_hdl.S` file; the panic handler interrupt is implemented there.

### 4.16.2 Notes

- Do not call C code from a high-priority interrupt; as these interrupts are run from a critical section, this can cause the target to crash. Note that although the panic handler interrupt does call normal C code, this exception is allowed due to the fact that this handler never returns (i.e., the application does not continue to run after the panic handler), so breaking C code execution flow is not a problem.

- Make sure your assembly code gets linked in. Indeed, as the free-to-use symbols are declared as weak, the linker may discard the file containing the symbol. This happens if the only symbol defined, or used from the user file is the `xt_*` free-to-use symbol. To avoid this, in the assembly file containing the `xt_*` symbol, define another symbol, like:

```
.global ld_include_my_isr_file
ld_include_my_isr_file:
```

Here it is called `ld_include_my_isr_file` but can have any name, as long as it is not defined anywhere else in the project.

Then, in the component `CMakeLists.txt`, add this name as an unresolved symbol to the `ld` command line arguments:

```
target_link_libraries(${COMPONENT_TARGET} "-u ld_include_my_isr_file")
```

This will ensure the linker to always includes the file defining `ld_include_my_isr_file`, so that the ISR is always linked.

  - High-priority 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. ESP-IDF does not support this yet.
  - To check Xtensa instruction set architecture (ISA), please refer to [Xtensa ISA Summary](#).

### 4.17 JTAG Debugging

This document provides a guide to installing OpenOCD for ESP32-S2 and debugging using GDB.

**Note:** You can also debug your ESP32-S2 without needing to setup JTAG or OpenOCD by using `idf.py monitor`. See: [IDF Monitor](#) and `CONFIG_ESP_SYSTEM_GDBSTUB_RUNTIME`.

The document is structured as follows:

- **Introduction** Introduction to the purpose of this guide.
- **How it Works?** Description how ESP32-S2, JTAG interface, OpenOCD and GDB are interconnected and working together to enable debugging of ESP32-S2.
- **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-S2 Target** Configuration of OpenOCD software and setting up of JTAG adapter hardware, which together make up the 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 to JTAG debugging of ESP32-S2 with OpenOCD and GDB.

4.17.1 Introduction

Espressif has ported OpenOCD to support the ESP32-S2 processor and the multi-core FreeRTOS (which is the foundation of most ESP32-S2 apps). Additionally, some extra tools have been written to provide extra features that OpenOCD does not support natively.

This document provides a guide to installing OpenOCD for ESP32-S2 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 LTS. 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.

4.17.2 How it Works?

The key software and hardware components that perform debugging of ESP32-S2 with OpenOCD over JTAG (Joint Test Action Group) interface is presented in the diagram below under the “Debugging With JTAG” label. These components include xtensa-esp32s2-elf-gdb debugger, OpenOCD on chip debugger, and the JTAG adapter connected to ESP32-S2 target.

Likewise, the “Application Loading and Monitoring” label indicates the key software and hardware components that allow an application to be compiled, built, and flashed to ESP32-S2, as well as to provide means to monitor diagnostic messages from ESP32-S2.
Debugging With JTAG and Application Loading and Monitoring is integrated under the Eclipse IDE in order to provide a quick and easy transition between writing/compiling/loading/debugging code. The Eclipse IDE (and the integrated debugging software) is available for Windows, Linux and macOS platforms. Depending on user preferences, both the debugger and `idf.py build` can also be used directly from terminal/command line, instead of Eclipse.

If the ESP32-S2-Kaluga-1 is used, then connection from PC to ESP32-S2 is done effectively with a single USB cable. This is made possible by the FT2232H chip, which provides two USB channels, one for JTAG and the other for UART connection.

### 4.17.3 Selecting JTAG Adapter

The quickest and most convenient way to start with JTAG debugging is by using ESP32-S2-Kaluga-1. Each version of this development board has JTAG interface already built in. No need for an external JTAG adapter and extra wiring/cable to connect JTAG to ESP32-S2. ESP32-S2-Kaluga-1 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-S2 as well as with the OpenOCD software. The JTAG port on the ESP32-S2 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 [https://openocd.org/doc/html/Debug-Adapter-Hardware.html](https://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-S2 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-S2 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-S2, although for now, support in OpenOCD for that line is pretty minimal.

ESP-Prog is an example for using an external board for debugging by connecting it to the JTAG pins of ESP32-S2.

### 4.17.4 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 https://openocd.org/doc/doxygen/bugs.html
```

You may also verify that OpenOCD knows where its configuration scripts are located by printing the value of `OPENOCDSCRIPTS` environment variable, by typing `echo $OPENOCDSCRIPTS` (for Linux and macOS) or `echo %OPENOCDSCRIPTS%` (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.
4.17.5 Configuring ESP32-S2 Target

Once OpenOCD is installed, you can proceed to configuring the ESP32-S2 target (i.e ESP32-S2 board with JTAG interface). Configuring the target is split into the following three steps:

- Configure and Connect JTAG Interface
- Run OpenOCD
- Upload Application for Debugging

Configure and Connect JTAG Interface

This step depends on the JTAG and ESP32-S2 board you are using (see the two cases described below).

Configure ESP32-S2-Kaluga-1 JTAG Interface

All versions of ESP32-S2-Kaluga-1 boards have built-in JTAG functionality. Putting it to work requires setting jumpers or DIP switches to enable JTAG functionality, and configuring USB drivers. Please refer to step by step instructions below.

Configure Hardware

- Out of the box, ESP32-S2-Kaluga-1 doesn’t need any additional hardware configuration for JTAG debugging. However if you are experiencing issues, check that the “JTAG” DIP switches (SW5 in the schematic) labelled TCK, TDO, TDI, TMS are in “ON” position.
- Verify if ESP32-S2 pins used for JTAG communication are not connected to some other h/w that may disturb JTAG operation:

<table>
<thead>
<tr>
<th>ESP32-S2 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTDO / GPIO40</td>
<td>TDO</td>
</tr>
<tr>
<td>MTDI / GPIO41</td>
<td>TDI</td>
</tr>
<tr>
<td>MTCK / GPIO39</td>
<td>TCK</td>
</tr>
<tr>
<td>MTMS / GPIO42</td>
<td>TMS</td>
</tr>
</tbody>
</table>

Configure USB Drivers

Install and configure USB drivers, so OpenOCD is able to communicate with JTAG interface on ESP32-S2-Kaluga-1 board as well as with UART interface used to upload application for flash. Follow steps below specific to your operating system.

Note: ESP32-S2-Kaluga-1 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 ESP32-S2-Kaluga-1 to the computer. Switch the ESP32-S2-Kaluga-1 on.
2. Wait until USB ports of ESP32-S2-Kaluga-1 are recognized by Windows and drives are installed. If they do not install automatically, then download them from https://ftdichip.com/drivers/d2xx-drivers/ and install manually.
3. Download Zadig tool (Zadig_X.X.exe) from https://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 ESP32-S2-Kaluga-1 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.
6. The first device (Dual RS232-HS (Interface 0)) is connected to the JTAG port of the ESP32-S2. 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-S2’s serial port (UART) used for upload of application to ESP32-S2’s flash.

Now ESP32-S2-Kaluga-1’s JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section *Run OpenOCD*.

**Linux**

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. To set up access permissions to USB devices supported by OpenOCD, copy the `udev rules file` into the `/etc/udev/rules.d` directory.
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-S2’s serial port (UART) used for upload of application to ESP32-S2’s flash.

Now ESP32-S2-Kaluga-1’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 FT2232 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 FT2232 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 does not load itself for channel A of FT2232 chip, which is the channel used for JTAG on ESP32-S2-Kaluga-1.

Manually unloading the driver

1. Install FTDI driver from https://ftdichip.com/drivers/vcp-drivers/
2. Connect USB cable to the ESP32-S2-Kaluga-1.
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:

- macOS < 10.15:

```
sudo kextunload -b com.apple.driver.AppleUSBFTDI
```

- macOS 10.15:

```
sudo kextunload -b com.apple.DriverKit-AppleUSBFTDI
```

**Warning:** Attempting to use serial over the wrong channel with the FTDI driver will cause a kernel panic. The ESP-WROVER-KIT uses channel A for JTAG and channel B for serial.

4. Run OpenOCD:

```
openocd -f board/esp32s2-kaluga-1.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 ESP32-S2-Kaluga-1 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:
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 Interfaces

For guidance about which JTAG interface to select when using OpenOCD with ESP32-S2, refer to the section Selecting JTAG Adapter. Then follow the configuration steps below to get it working.

Configure Hardware

1. Identify all pins/signals on JTAG interface and ESP32-S2 board that should be connected to establish communication.

   Table 4: ESP32-S2 pins and JTAG signals

<table>
<thead>
<tr>
<th>ESP32-S2 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTDO / GPIO40</td>
<td>TDO</td>
</tr>
<tr>
<td>MTDI / GPIO41</td>
<td>TDI</td>
</tr>
<tr>
<td>MTCK / GPIO39</td>
<td>TCK</td>
</tr>
<tr>
<td>MTMS / GPIO42</td>
<td>TMS</td>
</tr>
</tbody>
</table>

2. Verify if ESP32-S2 pins used for JTAG communication are not connected to some other hardware that may disturb JTAG operation.
3. Connect identified pin/signals of ESP32-S2 and JTAG interface.

Configure Drivers You may need to install driver software to make JTAG work with computer. Refer to documentation of your JTAG adapter for related details.

On Linux, adding OpenOCD udev rules is required and is done by copying the udev rules file into the `/etc/udev/rules.d` directory.
Connect

Connect JTAG interface to the computer. Power on ESP32-S2 and JTAG interface boards. Check if the JTAG interface is visible on the 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/esp32s2-kaluga-1.cfg
```

Note: The files provided after `-f` above are specific for ESP32-S2-Kaluga-1 board. You may need to provide different files depending on the hardware that is used. For guidance see Configuration of OpenOCD for Specific Target.

You should now see similar output (this log is for ESP32-S2-Kaluga-1 board):

```
user-name@computer-name:~/esp/esp-idf$ openocd -f board/esp32s2-kaluga-1.cfg
Open On-Chip Debugger v0.10.0-esp32-20200420 (2020-04-20-16:15)
Licensed under GNU GPL v2
For bug reports, read https://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: esp32s2.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),
part: 0x2003, ver: 0x1)
Info : esp32s2: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
Info : esp32s2: Core was reset (pwrstat=0x5F, after clear 0x0F).
```

- If there is an error indicating permission problems, please see section on “Permissions delegation” in the OpenOCD README file located in the `/esp/openocd-esp32` directory.
- In case there is an error in finding the configuration files, e.g., Can't find board/esp32s2-kaluga-1.cfg, check if the OPENOCD_SCRIPTS environment variable is set correctly. This variable is used by OpenOCD to look for the files specified after the `-f` option. See Setup of OpenOCD section for details. Also check if the file is indeed under the provided path.
- If you see JTAG errors (e.g., ...all ones or ...all zeroes), please check your JTAG connections, whether other signals are connected to JTAG besides ESP32-S2’s pins, and see if everything is powered on correctly.

Upload Application for Debugging

Build and upload your application to ESP32-S2 as usual, see Step 5. First Steps on ESP-IDF.

Another option is to write application image to flash using OpenOCD via JTAG with commands like this:

```
openocd -f board/esp32s2-kaluga-1.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] [compress] [en-
crypt]
```
• 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.
• compress - Optional. Compress image file before programming.
• encrypt - Optional. Encrypt binary before writing to flash. Same functionality with idf.py encrypted-flash

You are now ready to start application debugging. Follow the steps described in the section below.

4.17.6 Launching Debugger

The toolchain for ESP32-S2 features GNU Debugger, in short GDB. It is available with other toolchain programs under filename: xtensa-esp32s2-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.

The options of using debugger are discussed under links below.

- Eclipse
- Command Line
- Configuration for Visual Studio Code Debug

It is recommended to first check if debugger works from Command Line and then move to using Eclipse.

4.17.7 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.

Note: Debugging FreeRTOS Objects is currently only available for command line debugging.

Before proceeding to examples, set up your ESP32-S2 target and load it with get-started/blink.

4.17.8 Building OpenOCD from Sources

Please refer to separate documents listed below, that describe build process.

Building OpenOCD from Sources for Windows
Chapter 4. API Guides

Note: This document outlines how to build a binary of OpenOCD from its source files instead of downloading the pre-built binary. For a quick setup, users can download a pre-built binary of OpenOCD from Espressif GitHub instead of compiling it themselves (see Setup of OpenOCD for more details).

Note: All code snippets in this document are assumed to be running in an MSYS2 shell with the 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-winthreads-git
p7zip
```

Download Sources of OpenOCD

The sources for the ESP32-S2-enabled variant of OpenOCD are available from Espressif’s GitHub under https://github.com/espressif/openocd-esp32. These source files can be pulled via Git using 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

The libusb library is also required when building OpenOCD. The following commands will download a particular release of libusb and unpack it to the current directory.

```
wget https://github.com/libusb/libusb/releases/download/v1.0.22/libusb-1.0.22.7z
7z x -olibusb ./libusb-1.0.22.7z
```

We now need to export the following variables such that the libusb library gets linked into the OpenOCD build.

```
export CPPFLAGS="$CPPFLAGS -I$(PWD)/libusb/include/libusb-1.0.22"
export LDFLAGS="$LDFLAGS -L$(PWD)/libusb/MinGW32/.libs/dll"
```

Build OpenOCD

The following commands will configure OpenOCD then build it.

```
cd ~/esp/openocd-esp32
export CPPFLAGS="$CPPFLAGS -D__USE_MINGW_ANSI_STDIO=1 -Wno-error"; export CFLAGS="-s$CFLAGS -Wno-error"
./bootstrap
./configure --disable-doxygen-pdf --disable-ftdi --enable-jlink --enable-ulink --build=1686-w64-mingw32 --host=1686-w64-mingw32
make
cp ../libusb/MinGW32/dll/libusb-1.0.dll ./src
cp /opt/i686-w64-mingw32/bin/libwinpthread-1.dll ./src
```

Once the build is completed, the OpenOCD binary will be placed in ~/esp/openocd-esp32/src/.

You can then optionally call make install. This will copy the OpenOCD binary to a user specified location.

- This location can be specified when OpenOCD is configured, or by setting export DESTDIR="/custom/install/dir" before calling make install.
Chapter 4. API Guides

- If you have an existing OpenOCD (from e.g., another development platform), you may want to skip this call as your existing OpenOCD 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.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** For greater convenience, all of commands called throughout the OpenOCD build process have been listed in the code snippet below. Users can copy this code snippet into a shell script then execute it:

```bash
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-\ncross-winthreads-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

echo "CPPFLAGS="$CPPFLAGS"-I${PWD}/libusb/include/libusb-1.0"; echo "LDFLAGS="
"-L${PWD}/libusb/MinGW32/.libs/dll"

export CPPFLAGS="SCPPFLAGS=-D__USE_MINGW_ANSI_STDIO=1 -Wno-error"; export CFLAGS="
"-DCFLAGS="-Wno-error"

cd ~/esp/openocd-esp32

./bootstrap

./configure --disable-doxygen-pdf --disable-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-S2 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-S2-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.
Chapter 4. API Guides

Next Steps  To carry on with debugging environment setup, proceed to section Configuring ESP32-S2 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-S2-enabled variant of OpenOCD are available from Espressif GitHub under https://github.com/espressif/openocd-esp32. To download the sources, use the following commands:

```bash
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:

```bash
brew install automake libtool libusb wget gcc@4.9 pkg-config
```

Build OpenOCD  Proceed with configuring and building OpenOCD:

```bash
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.
- Error Unknown command 'raggedright' may indicate that the required version of texinfo was not installed on your computer or installed but was not linked to your PATH. To resolve this issue make sure texinfo is installed and PATH is adjusted prior to the ./bootstrap by running:

```
brew install texinfo
export PATH=/usr/local/opt/texinfo/bin:$PATH
```

- 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-S2 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:

```bash
cd ~/esp/openocd-esp32
export OPENOCD_SCRIPTS=$PWD/tcl
```

For Windows:

```bash
cd %USERPROFILE%\esp\openocd-esp32
set "OPENOCD_SCRIPTS=%CD%\tcl"
```

Example of invoking OpenOCD build locally from sources, for Linux and macOS:

```bash
src/openocd -f board/esp32s2-kaluga-1.cfg
```

and Windows:

```bash
src\openocd -f board/esp32s2-kaluga-1.cfg
```

### 4.17.9 Tips and Quirks

This section provides collection of links to all tips and quirks referred to from various parts of this guide.

#### 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-S2 debugger supports 2 hardware implemented breakpoints and 64 software ones. Hardware breakpoints are implemented by ESP32-S2 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 cannot 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-S2 also supports 2 watchpoints, so 2 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 last 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-S2 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:

```
esp 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/esp32s2-kaluga-1.cfg -c "init; halt; esp 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 ahead in the code to bypass the subroutine calls. If all 2 breakpoints are already set, this functionality will not work. If this is the case, delete breakpoints to have one “spare”. With all 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_ESP_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.

GDB has a Python extension for FreeRTOS support. ESP-IDF automatically loads this module into GDB with the `idf.py gdb` command when the system requirements are met. See more details in Debugging FreeRTOS Objects.

**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 MHz or 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 MHz or 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 available hardware watchpoints supported by the chip, 2 for ESP32-S2. For more information see https://sourceware.org/gdb/onlinedocs/gdb/Remote-Configuration.html.
- `mon reset halt` — reset the chip and keep the CPUs halted
- `maintenance flush register-cache` — `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 `maintenance flush register-cache` 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 There are several kinds of OpenOCD configuration files (*.cfg). All configuration files are located in subdirectories of `share/openocd/scripts` directory of OpenOCD distribution (or `tcl/scripts` directory of the source repository). For the purposes of this guide, the most important ones are `board`, `interface` and `target`.

- `interface` configuration files describe the JTAG adapter. Examples of JTAG adapters are ESP-Prog and J-Link.
- `target` configuration files describe specific chips, or in some cases, modules.
- `board` configuration files are provided for development boards with a built-in JTAG adapter. Such files include an `interface` configuration file to choose the adapter, and `target` configuration file to choose the chip/module.

The following configuration files are available for ESP32-S2:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>board/esp32s2-kaluga-1.cfg</code></td>
<td>Board configuration file for ESP32-S2-Kaluga-1, includes target and adapter configuration.</td>
</tr>
<tr>
<td><code>target/esp32s2.cfg</code></td>
<td>ESP32-S2 target configuration file. Can be used together with one of the interface/configuration files.</td>
</tr>
<tr>
<td><code>interface/ftdi/esp32s2_kaluga_v1.cfg</code></td>
<td>JTAG adapter configuration file for ESP32-S2-Kaluga-1 board.</td>
</tr>
<tr>
<td><code>interface/ftdi/esp32_devkitj_v1.cfg</code></td>
<td>JTAG adapter configuration file for ESP-Prog boards.</td>
</tr>
</tbody>
</table>

If you are using one of the boards which have a pre-defined configuration file, you only need to pass one `-f` argument to OpenOCD, specifying that file.

If you are using a board not listed here, you need to specify both the interface configuration file and target configuration file.

Custom Configuration Files OpenOCD configuration files are written in TCL, and include a variety of choices for customization and scripting. This can be useful for non-standard debugging situations. Please refer to OpenOCD Manual for the TCL scripting reference.

OpenOCD Configuration Variables The following variables can be optionally set before including the ESP-specific target configuration file. This can be done either in a custom configuration file, or from the command line.

The syntax for setting a variable in TCL is:
To set a variable from the command line (replace the name of .cfg file with the correct file for your board):

```
openocd -c 'set VARIABLE_NAME value' -f board/esp-xxxxx-kit.cfg
```

It is important to set the variable before including the ESP-specific configuration file, otherwise the variable will not have effect. You can set multiple variables by repeating the `-c` option.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP_RTOS</td>
<td>Set to <code>none</code> to disable RTOS support. In this case, thread list will not be available in GDB. Can be useful when debugging FreeRTOS itself, and stepping through the scheduler code.</td>
</tr>
<tr>
<td>ESP_FLASH_SIZE</td>
<td>Set to 0 to disable Flash breakpoints support.</td>
</tr>
<tr>
<td>ESP_SEMIHOST_BASED</td>
<td>Set to the path (on the host) which will be the default directory for semihosting functions.</td>
</tr>
</tbody>
</table>

### How Debugger Resets ESP32-S2?

The board can be reset by entering `mon reset` or `mon reset halt` into GDB.

### Can JTAG Pins Be Used for Other Purposes?

Operation of JTAG may be disturbed, if some other hardware is connected to JTAG pins besides ESP32-S2 module and JTAG adapter. ESP32-S2 JTAG is using the following pins:

<table>
<thead>
<tr>
<th>ESP32-S2 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTD0 / GPIO40</td>
<td>TDO</td>
</tr>
<tr>
<td>MTD1 / GPIO41</td>
<td>TDI</td>
</tr>
<tr>
<td>MTCK / GPIO39</td>
<td>TCK</td>
</tr>
<tr>
<td>MTMS / GPIO42</td>
<td>TMS</td>
</tr>
</tbody>
</table>

JTAG communication will likely fail, if configuration of JTAG pins is changed by a user application. If OpenOCD initializes correctly (detects all the CPU cores in the SOC), but loses sync and spews out a lot of DTR/DIR errors when the program is running, it is likely that the application reconfigures the JTAG pins to something else, or the user forgot to connect Vtart to a JTAG adapter that requires it.

### 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, see [OpenOCD Configuration Variables](#).
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.

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, and the chip/module being debugged.
   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/esp32s2-kaluga-1.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/esp32s2-kaluga-1.cfg 2>&1 | tee openocd.log
   ```

   Debugger:

   ```
   xtensa-esp32s2-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.

4.17.10 Related Documents

Using Debugger

This section covers the steps to configure and run a debugger using various methods, including:

- **Eclipse**
- **Command Line**
- **Idf.py Debug Targets**

For how to run a debugger from VS Code, see Configuration for Visual Studio Code Debug.

**Eclipse**

**Note:** It is recommended to first check if debugger works using Idf.py Debug Targets or from Command Line and then move to using Eclipse.

Eclipse is an integrated development environment (IDE) that provides a powerful set of tools for developing and debugging software applications. For ESP-IDF applications, IDF Eclipse plugin provides two ways of debugging:

1. **ESP-IDF GDB OpenOCD Debugging**
2. **GDB Hardware Debugging**
By default, Eclipse supports OpenOCD Debugging via the GDB Hardware Debugging plugin, which requires starting the OpenOCD server from the command line and configuring the GDB client from Eclipse to start with the debugging. This approach can be time-consuming and error-prone.

To make the debugging process easier, the IDF Eclipse plugin has a customized ESP-IDF GDB OpenOCD Debugging functionality. This functionality supports configuring the OpenOCD server and GDB client from within Eclipse. All the required configuration parameters will be pre-filled by the plugin, and you can start debugging with just a click of a button.

Therefore, it is recommended to use the ESP-IDF GDB OpenOCD Debugging via the IDF Eclipse plugin.

**GDB Hardware Debugging**

**Note:** This approach is recommended only if you are unable to debug using ESP-IDF GDB OpenOCD Debugging for some reason.

To install the GDB Hardware Debugging plugin, open Eclipse and select Help > Install New Software. After installation is complete, follow these steps to configure the debugging session. Please note that some configuration parameters are generic, while others 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 Eclipse Plugin. 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 left pane of the window, 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 the Browse button and select the blink project.
4. In the next line under C/C++ Application:. press the Browse button and select the blink.elf file. If blink.elf is not there, it is likely that this project has not been built yet. Refer to the Eclipse Plugin for instructions.
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.
6. Click the Debugger tab. In field GDB Command, enter xtensa-esp32s2-elf-gdb to invoke the debugger.
7. Change the default configuration of the 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 that requires changing the default configuration is Startup. Under Initialization Commands uncheck Reset and Delay (seconds) and Halt. Then, in the entry field below, enter the following lines:
   ```
   mon reset halt
   maintenance flush register-cache
   set remote hardware-watchpoint-limit 2
   ```

**Note:** To automatically update the image in the flash before starting a new debug session, add the following command lines to the beginning of the 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. Uncheck the Load image option under Load Image and Symbols.
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:. Check out 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.

Configuration described in points 8 - 11 is shown below.
Fig. 28: Configuration of GDB Hardware Debugging - Main tab
Fig. 29: Configuration of GDB Hardware Debugging - Debugger tab
Fig. 30: Configuration of GDB Hardware Debugging - Startup tab
If the Startup sequence looks convoluted and respective Initialization Commands are unclear, check What Is the Meaning of Debugger’s Startup Commands? for additional explanation.

12. If you have completed the Configuring ESP32-S2 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-S2 Target and return here to start debugging.

Once all configuration steps 1-12 are satisfied, the new Eclipse perspective called “Debug” will open, as shown in the example picture below.

![Debug Perspective in Eclipse](image)

If you are not quite sure how to use GDB, check Eclipse example debugging session in section Debugging Examples.

**Command Line**

1. Begin by completing the steps described under Configuring ESP32-S2 Target. This is prerequisite to start a debugging session.

2. Open a new terminal session and go to the directory that contains the project for debugging, e.g.,

   ```
   cd ~/esp/blink
   ```

3. When launching a debugger, you will need to provide a couple of configuration parameters and commands. Instead of entering them one by one in the command line, create a configuration file and name it `gdbinit`:

   ```
   target remote :3333
   set remote hardware-watchpoint-limit 2
   mon reset halt
   maintenance flush register-cache
   thb app_main
   c
   ```
Chapter 4. API Guides

Save this file in the current directory.
For more details on what is inside `gdbinit` file, see What Is the Meaning of Debugger’s Startup Commands?

4. Now you are ready to launch GDB. Type the following in terminal:

```bash
xtensa-esp32s2-elf-gdb -x gdbinit build/blink.elf
```

5. If the previous steps have been done correctly, you will see a similar log concluded with the (gdb) prompt:

```plaintext
user-name@computer-name:/~/esp/blink$ xtensa-esp32s2-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- → esp32s2-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
Find the GDB manual and other documentation resources online at:
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/esp32s2/./freertos_hooks.c:52
52          asm("waiti 0");
JTAG tap: esp32s2.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),... → part: 0x2003, ver: 0x1)
JTAG tap: esp32s2.slave tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),... → part: 0x2003, ver: 0x1)
esp32s2: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
esp32s2: Core was reset (pwrstat=0x5F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x50000400 (active) APP_CPU: PC=0x00000000
esp32s2: target state: halted
esp32s2: Core was reset (pwrstat=0x1F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x04000400 (active) APP_CPU: PC=0x04000400
esp32s2: 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 107341316]
[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
43           xTaskCreate(&blink_task, "blink_task", 512, NULL, 5, NULL);
(gdb)
```

Note the third-to-last line, which 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 sure how to use GDB, check Command Line example debugging session in section Debugging Examples.
**Idf.py Debug Targets**  It is also possible to execute the described debugging tools conveniently from idf.py. These commands are supported:

1. **idf.py openocd**
   - Runs OpenOCD in a console with configuration defined in the environment or via command line. It uses default script directory defined as OPENOCD_SCRIPTS environmental variable, which is automatically added from an Export script (export.sh or export.bat). It is possible to override the script location using command line argument `--openocd-scripts`.
   - To configure the JTAG configuration for the current board, please use the environmental variable OPENOCD_COMMANDS or `--openocd-commands` command line argument. If none of the above is defined, OpenOCD is started with `-f board/esp32s2-kaluga-1.cfg` board definition.

2. **idf.py gdb**
   - Starts the GDB the same way as the **Command Line**, but generates the initial GDB scripts referring to the current project elf file.

3. **idf.py gdbtui**
   - The same as 2, but starts the gdb with `tui` argument, allowing for a simple source code view.

4. **idf.py gdbgui**
   - Starts `gdbgui` debugger frontend enabling out-of-the-box debugging in a browser window. To enable this option, run the install script with the `--enable-gdbgui` argument, e.g., `install.sh --enable-gdbgui`.
   - You can combine these debugging actions on a single command line, allowing for convenient setup of blocking and non-blocking actions in one step. `idf.py` implements a simple logic to move the background actions (such as openocd) to the beginning and the interactive ones (such as gdb, monitor) to the end of the action list.
   - An example of a very useful combination is:
     
     ```
     idf.py openocd gdbgui monitor
     ```
   - The above command runs OpenOCD in the background, starts `gdbgui` to open a browser window with active debugger frontend and opens a serial monitor in the active console.

### 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()`.

#### 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.

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.
Fig. 32: Debug Perspective in Eclipse
Fig. 33: Target halted during debugging
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.

![Image showing the call stack of a program](image)

**Fig. 34: Navigate through the call stack**

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/periherals. 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. 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.

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 “Thread #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.
Fig. 35: Setting a breakpoint
Fig. 36: Three breakpoints are set / maximum two are allowed
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 thread 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.

**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.

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.
Fig. 38: Stepping through the code with "Step Over (F6)"
Fig. 39: Stepping through the code with "Step Into (F5)"
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, see ESP32-S2 Technical Reference Manual > IO MUX and GPIO Matrix (GPIO, IO_MUX) [PDF].

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.

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 tasks 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.

![Watching program variable “i”](image)

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()`:
Fig. 43: Setting a conditional breakpoint
Examples in This Section

1. Navigating Through the Code, Call Stack and Threads
2. Setting and Clearing Breakpoints
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
8. Debugging FreeRTOS Objects

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 \texttt{l} or \texttt{list}, and debugger will show couple of lines of code around the halt point (line 43 of code in file \texttt{blink.c})

```plaintext
(gdb) l
38 }
39 }
40
41 void app_main()
42 {
43 xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5, _
44 };
(gdb)
```

Check how code listing works by entering, e.g., \texttt{l 30, 40} to see particular range of lines of code.

You can use \texttt{bt} or \texttt{backtrace} to see what function calls lead up to this code:

```plaintext
(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/
    esp32s2/./cpu_start.c:339
(gdb)
```

Line #0 of output provides the last function call before the application halted, i.e., \texttt{app_main ()} we have listed previously. The \texttt{app_main ()} was in turn called by function \texttt{main_task} from line 339 of code located in file \texttt{cpu_start.c}.

To get to the context of \texttt{main_task} in file \texttt{cpu_start.c}, enter frame \texttt{N}, where \texttt{N = 1}, because the \texttt{main_task} is listed under \texttt{#1}:

```plaintext
(gdb) frame 1
#1 0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
    esp32s2/./cpu_start.c:339
339    app_main();
(gdb)
```

Enter \texttt{l} and this will reveal the piece of code that called \texttt{app_main()} (in line 339):
By listing some lines before, you will see the function name `main_task` we have been looking for:

```
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 }
```

To see the other code, enter `i threads`. This will show the list of threads running on target:

```
Id Target Id  Frame
  1 Thread 1073411772 (main : Running) app_main () at /home/user-name/esp/blink/main/./blink.c:43
  2 Thread 1073413512 (IDLE) prvIdleTask (pvParameters=0x0) at /home/user-name/esp/blink/main/./blink.c:43
  3 Thread 1073413156 (IDLE) prvIdleTask (pvParameters=0x0) at /home/user-name/esp/blink/main/./blink.c:43
  4 Thread 1073432224 (dport) dport_access_init_core (arg=0x0) at /home/user-name/esp/esp-idf/components/esp32s2/./dport_access.c:150
  5 Thread 1073411208 (ipc1 : Running) 0x400bf6ea in ?? ()
  6 Thread 1073431096 (Tmr Svc) prvTimerTask (pvParameters=0x0) at /home/user-name/esp/esp-idf/components/freertos/.timers.c:445
  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
  8 Thread 1073411336 (dport) 0x400d0848 in dport_access_init_core (arg=0x0) at /home/user-name/esp/esp-idf/components/esp32s2/./dport_access.c:170
```

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 5:
Then check the backtrace:

```plaintext
(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
#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` 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:

```plaintext
(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 new 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:

```plaintext
(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
    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
    gpio_set_level(BLINK_GPIO, 1);
```

(continues on next page)
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
    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 breakpoints 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 1073413352]
Program received signal SIGINT, Interrupt.
[Switching to Thread 1073413512]
0x400d0c00 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/components/esp32s2/./freertos_hooks.c:52
    asm("waiti 0");
```

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.

**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) break
(breakpoint 3) (gdb)
```
(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/./
→blinky.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/esptagpio/src/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, see ESP32-S2 Technical Reference Manual > IO MUX and GPIO Matrix (GPIO, IO_MUX) [PDF].
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
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 your 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
```

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)
```
Continuing.

Target halted. PRO_CPU: PC=0x400DB751 (active)    APP_CPU: PC=0x400D0811

Program received signal SIGTRAP, Trace/breakpoint trap.

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./

34 gpio_set_level(BLINK_GPIO, 0);

(gdb)

Resume application couple more times so \textit{i} gets incremented. Now you can enter \texttt{print i} (in short \texttt{p i}) to check the current value of \textit{i}:

\begin{verbatim}
(gdb) p i
$1 = 3
(gdb)
\end{verbatim}

To modify the value of \textit{i} use \texttt{set} command as below (you can then print it out to check if it has been indeed changed):

\begin{verbatim}
(gdb) set var i = 0
(gdb) p i
$3 = 0
(gdb)
\end{verbatim}

You may have up to two watchpoints, see \textit{Breakpoints and Watchpoints Available}.

\subsection*{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:

\begin{verbatim}
(gdb) break blink.c:34 if (i == 2)
Breakpoint 3 at 0x400db753: file /home/user-name/esp/blink/main/./blink.c, line 34.
(gdb)
\end{verbatim}

Above command sets conditional breakpoint to halt program execution in line 34 of \texttt{blink.c} if \texttt{i == 2}.

If current value of \textit{i} is less than 2 and program is resumed, it will blink LED in a loop until condition \texttt{i == 2} gets true and then finally halt:

\begin{verbatim}
(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/./

34 gpio_set_level(BLINK_GPIO, 0);
(gdb)
\end{verbatim}

\subsection*{Debugging FreeRTOS Objects}

This part might be interesting when you are debugging FreeRTOS tasks interactions.

Users that need to use the FreeRTOS task interactions can use the GDB \texttt{freertos} command. The \texttt{freertos} command is not native to GDB and comes from the \texttt{freertos-gdb} Python extension module. The \texttt{freertos} command contains a series of sub-commands as demonstrated in the code snippet:
Chapter 4. API Guides

(gdb) freertos
"freertos" must be followed by the name of a subcommand.
List of freertos subcommands:
freertos queue -- Generate a print out of the current queues info.
freertos semaphore -- Generate a print out of the current semaphores info.
freertos task -- Generate a print out of the current tasks and their states.
freertos timer -- Generate a print out of the current timers info.

For a more detailed description of this extension, please refer to https://pypi.org/project/freertos-gdb.

**Note:** The freertos-gdb Python module is included as a Python package requirement by ESP-IDF, thus should be automatically installed (see Step 3. Set up the Tools for more details).

The FreeRTOS extension automatically loads in case GDB is executed with command via idf.py gdb. Otherwise, the module could be enabled via the python import freertos_gdb command inside GDB.

Users only need to have Python 3.6 (or above) that contains a Python shared library.

**Obtaining Help on Commands** Commands presented so far should provide are very basis 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 particular command, being at (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 top level list of command classes, to aid you drilling down to more details. Optionally refer to available GDB cheat sheets, for instance https://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 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$

• Using Debugger
• Debugging Examples
• Tips and Quirks
• Application Level Tracing Library
• Introduction to ESP-Prog Board

4.18 Linker Script Generation
4.18.1 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.
- executable code in IRAM so that it can be ran while cache is disabled.
- code in RTC memory for use in a wake stub.
- code in RTC memory for use by 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.

4.18.2 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
      ├── Kconfig
      │    └── 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 is 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:

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:

```plaintext
[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:

```plaintext
[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:

```plaintext
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:

```plaintext
[mapping:my_component]
archive: libmy_component.a
entries:
  * (noflash)
```

Similarly, this places the entire component in RTC memory:

```plaintext
[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:

```plaintext
[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:

```plaintext
[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:

```plaintext
[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 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.

### 4.18.3 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.
  - For sections and scheme, the only supported key is entries
  - For mappings, both archive and entries are supported.

**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 kconfiglib package and adheres to its required syntax and limitations. Supported operators are as follows:

- **comparison**
  - LessThan <
  - LessThanOrEqualTo <=

Espressif Systems  1943  Release master
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:

```plaintext
# 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
```

```plaintext
# Entire fragment definition is dependent on config
if CONDITION = y:
  [type:name]
  key_1:
    value_1
  key_2:
    value_a
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.

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.

```plaintext
[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 will be generated for the target flash_text.

* (.literal .literal.* .text .text.*)

These catch-all rules then effectively serve as fallback rules for those whose mappings were not specified.

The default scheme is defined in esp_system/app.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)
```

Aside from the entity and scheme, flags can also be specified in an entry. The following flags are supported (note: <> = argument name, [] = optional):

1. ALIGN(<alignment>[, pre, post])
   Align the placement by the amount specified in alignment. Generates

2. SORT([<sort_by_first>, <sort_by_second>])
   Emits SORT_BY_NAME, SORT_BY_ALIGNMENT, SORT_BY_INIT_PRIORITY or SORT in the input section description. Possible values for sort_by_first and sort_by_second are: name, alignment, init_priority. If both sort_by_first and sort_by_second are not specified, the input sections are sorted by name. If both are specified, then the nested sorting follows the same rules discussed in https://sourceware.org/binutils/docs/ld/Input-Section-Wildcards.html.

3. KEEP()
   Prevent the linker from discarding the placement by surrounding the input section description with KEEP command. See https://sourceware.org/binutils/docs/ld/Input-Section-Keep.html for more details.

4. SURROUND(<name>)
   Generate symbols before and after the placement. The generated symbols follow the naming __<name>_start and __<name>_end. For example, if name == sym1,

When adding flags, the specific section -> target in the scheme needs to be specified. For multiple section -> target, use a comma as a separator. For example,

```
# Notes:
# A. semicolon after entity-scheme
# B. comma before section2 -> target2
# C. section1 -> target1 and section2 -> target2 should be defined in entries of_.scheme1
```

(continues on next page)
Putting it all together, the following mapping fragment, for example,

```
[mapping:name]
archive: lib1.a
entries:
  obj1 (noflash);
  rodata -> dram0_data KEEP() SORT() ALIGN(8) SURROUND(my_sym)
```

generates an output on the linker script:

```
. = ALIGN(8)
_my_sym_start = ABSOLUTE(.
KEEP(lib1.a:obj1.*( SORT(.rodata) SORT(.rodata.*) )))
_my_sym_end = ABSOLUTE(.
```

Note that ALIGN and SURROUND, as mentioned in the flag descriptions, are order sensitive. Therefore, if for the same mapping fragment these two are switched, the following is generated instead:

```
_my_sym_start = ABSOLUTE(.
. = ALIGN(8)
KEEP(lib1.a:obj1.*( SORT(.rodata) SORT(.rodata.*) )))
_my_sym_end = ABSOLUTE(.
```

**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 `-fdata-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 */
```

(continues on next page)
Suppose the generator collected the fragment definitions below:

```plaintext
[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:

```plaintext
.iram0.text :
{
  /* Code marked as running out of IRAM */
  _iram_text_start = ABSOLUTE(.);

  /* Placement rules generated from the processed fragments, placed wherever 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.

  The linker script template currently used is esp_system/ld/esp32s2/sections.ld.in; the generated output
  script sections.ld is put under its build directory.

Espressif Systems

Submit Document Feedback
Migrate to ESP-IDF v5.0 Linker Script Fragment Files Grammar

The old grammar supported in ESP-IDF v3.x would be dropped in ESP-IDF v5.0. Here are a few notes on how to migrate properly:

1. Now indentation is enforced and improperly indented fragment files would generate a runtime parse exception. This was not enforced in the old version but previous documentation and examples demonstrate properly indented grammar.
2. Migrate the old condition entry to the `if...elif...else` structure for conditionals. You can refer to the earlier chapter for detailed grammar.
3. Mapping fragments now requires a name like other fragment types.

4.19 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.

4.19.1 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

**Warning:** When using any lwIP API other than the **BSD Sockets API**, please make sure that the API is thread-safe. To check if a given API call is thread-safe, enable the `CONFIG_LWIP_CHECK_THREAD_SAFETY` configuration option and run the application. This enables lwIP to assert the correct access of the TCP/IP core functionality. If the API is not accessed or locked properly from the appropriate lwIP FreeRTOS Task, the execution will be aborted. The general recommendation is to use the **ESP-NETIF** component to interact with lwIP.

Some common lwIP app APIs are supported indirectly by ESP-IDF:

- Dynamic Host Configuration Protocol (DHCP) Server & Client are supported indirectly via the **ESP-NETIF** functionality.
- Simple Network Time Protocol (SNTP) is also supported via the **ESP-NETIF**, or directly via the `lwip/include/apps/esp_snntp.h` functions, which also provide thread-safe API to `lwip/lwip/src/include/lwip/apps/snntp.h` functions, see also **SNTP Time Synchronization**.
- ICMP Ping is supported using a variation on the lwIP ping API, see **ICMP Echo**.
- ICMPv6 Ping, supported by lwIP's ICMPv6 Echo API, is used to test IPv6 network connectivity. For more information, see **protocols/sockets/icmp6_ping**.
- NetBIOS lookup is available using the standard lwIP API, `protocols/http_server/restful_server` has the 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.
- The PPP implementation in lwIP can be used to create PPPoS (PPP over serial) interface in ESP-IDF. Please refer to the documentation of the **ESP-NETIF** component to create and configure a PPP network interface, by means of the `ESP_NETIF_DEFAULT_PPP()` macro defined in `esp_netif/include/esp_netif_defaults.h`. Additional runtime settings are provided via `esp_netif/include/esp_netif_ppp.h`. PPPoS interfaces are typically
used to interact with NB-IoT/GSM/LTE modems. More application-level friendly API is supported by the `esp_modem` library, which uses this PPP lwIP module behind the scenes.

### 4.19.2 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 materials are 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`: this simplified example uses a TCP socket to send an HTTP request, but 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(), writev`: via Virtual Filesystem Component
- `recv(), recvmsg(), recvfrom()`
- `send(), sendmsg(), sendto()`
- `select()`: via Virtual Filesystem Component
- `poll()` 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 `ioctl`

**Note:** Some lwIP application sample code uses prefixed versions of BSD APIs, e.g., `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, socket error handling involves the following aspects:

- Detecting the error
- Getting the error reason code
- Handling the error according to the reason code

In lwIP, we have two different scenarios for 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 an exception descriptor indicating that the socket has an error. For more information, see **select() Errors**.

**Socket API Errors**  
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 does not 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:

```c
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**  
Error detection

- Socket error when `select()` has exception descriptor.

Get the error reason code

- If the `select()` indicates that the socket fails, we can not get the error reason code by accessing `errno`, instead we should call `getsockopt()` to get the failure reason code. Since `select()` has exception descriptor, the error code is not given to `errno`.

**Note:** The `getsockopt()` function has the following prototype: `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)`. Its purpose 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:

```c
int err;

if (select(sockfd + 1, NULL, NULL, &exfds, &tval) <= 0) {
  err = errno;
  return err;
} else {
  if (FD_ISSET(sockfd, &exfds)) {
(...continues on next page)
select() exception set using getsockopt()

```c
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 a more detailed list of standard POSIX/C error codes, please see newlib errno.h and the platform-specific extensions newlib/platform_include/errno.h.

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONNREFUSED</td>
<td>Connection refused</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>Address already in use</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>Software caused connection abort</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>Network is unreachable</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>Network interface is not configured</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>Connection timed out</td>
</tr>
<tr>
<td>EHOSTDOWN</td>
<td>Host is down</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>Host is unreachable</td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>Connection already in progress</td>
</tr>
<tr>
<td>EALREADY</td>
<td>Socket already connected</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>Destination address required</td>
</tr>
<tr>
<td>EPROTONOSUPPORT</td>
<td>Unknown protocol</td>
</tr>
</tbody>
</table>

Socket Options

The `getsockopt()` and `setsockopt()` functions allow getting and setting per-socket options respectively. 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, whose 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: 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
Chapter 4. API Guides

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, the 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 or 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 or write modes. These flags can only be read using F_GETFL, and cannot be set using F_SETFL. A TCP socket returns 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 the 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, ...).

4.19.3 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 firstly copying 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 part of the **unofficial** lwIP Application Developers Manual.

### 4.19.4 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`

### 4.19.5 IPv6 Support

Both IPv4 and IPv6 are supported in a dual-stack configuration and are enabled by default. Both IPv6 and IPv4 may be disabled separately if they are not needed, see Minimum RAM Usage.

IPv6 support is limited to **Stateless Autoconfiguration** only. **Stateful configuration** is not supported in ESP-IDF, nor in upstream lwIP.

IPv6 Address configuration is defined by means of these protocols or services:

- SLAAC IPv6 Stateless Address Autoconfiguration (RFC-2462)
- DHCPv6 Dynamic Host Configuration Protocol for IPv6 (RFC-8415)

None of these two types of address configuration is enabled by default, so the device uses only Link Local addresses or statically-defined addresses.

**Stateless Autoconfiguration Process**

To enable address autoconfiguration using the Router Advertisement protocol, please enable:

- `CONFIG_LWIP_IPV6_AUTOCONFIG`

This configuration option enables IPv6 autoconfiguration for all network interfaces, which differs from the upstream lwIP behavior, where the autoconfiguration needs to be explicitly enabled for each `netif` with `netif->ip6_autoconfig_enabled=1`.

**DHCPv6**

DHCPv6 in lwIP is very simple and supports only stateless configuration. It could be enabled using:

- `CONFIG_LWIP_IPV6_DHCP6`

Since the DHCPv6 works only in its stateless configuration, the **Stateless Autoconfiguration Process** has to be enabled as well via `CONFIG_LWIP_IPV6_AUTOCONFIG`.

Moreover, the DHCPv6 needs to be explicitly enabled from the application code using:

```
dhcp6_enable_stateless(netif);
```

**DNS Servers in IPv6 Autoconfiguration**

In order to autoconfigure DNS server(s), especially in IPv6-only networks, we have these two options:
Chapter 4. API Guides

- Recursive Domain Name System (DNS): this belongs to the Neighbor Discovery Protocol (NDP) and uses Stateless Autoconfiguration Process. The number of servers must be set CONFIG_LWIP_IPV6_RDNSS_MAX_DNS_SERVERS, this option is disabled by default, i.e., set to 0.
- DHCPv6 stateless configuration, uses DHCPv6 to configure DNS servers. Note that this configuration assumes IPv6 Router Advertisement Flags (RFC-5175) to be set to
  - Managed Address Configuration Flag = 0
  - Other Configuration Flag = 1

4.19.6 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 than the one that created it. The close() call blocks, 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 the stack so that when the task gets destroyed before unrolling the stack, the lwIP could still hold pointers to the deleted stack.

- **On-Demand Timers** lwIP IGMP and MLD6 feature 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 behavior is to set each timer on demand, so it is only enabled when an event is pending.

To return to the default lwIP behavior, which is always-on timers, disable CONFIG_LWIP_TIMERS_ONDEMAND.

- **lwIP Timers API** When not using Wi-Fi, the lwIP timer can be turned off via the API to reduce power consumption.

The following API functions are supported. For full details, see lwip/lwip/src/include/lwip/timeouts.h.

- sys_timeouts_init()
- sys_timeouts_deinit()

- **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, while normal lwIP is TCP-only.

- **IP Layer Features**
  - IPV4-source-based routing implementation is different
  - IPV4-mapped IPV6 addresses are supported

- **Customized lwIP Hooks** The original lwIP supports implementing custom compile-time modifications via LWIP_HOOK_FILENAME. This file is already used by the ESP-IDF port layer, but ESP-IDF users could still include and implement any custom additions via a header file defined by the macro ESP_IDF_LWIP_HOOK_FILENAME. Here is an example of adding a custom hook file to the build process, and the hook is called my_hook.h located in the project’s main folder:
### Limitations

Calling `send()` or `sendto()` repeatedly on a UDP socket may eventually fail with `errno` equal to `ENOMEM`. This failure occurs due to the limitations of buffer sizes in the lower-layer network interface drivers. If all driver transmit buffers are full, the UDP transmission will fail. For applications that transmit a high volume of UDP datagrams and aim to avoid any dropped datagrams by the sender, it is advisable to implement error code checking and employ a retransmission mechanism with a short delay.

Increasing the number of TX buffers in the **Wi-Fi** project configuration may also help.

### 4.19.7 Performance Optimization

TCP/IP performance is a complex subject, and performance can be optimized toward 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, it is suggested to apply 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 (18, `ESP_TASK_TCPIP_PRIO`). To optimize CPU utilization, consider assigning competing tasks to different cores or adjusting their priorities to lower values. For additional details on built-in task priorities, please refer to [Built-in Task Priorities](#).
- If using `select()` function with socket arguments only, disabling `CONFIG_VFS_SUPPORT_SELECT` will make `select()` calls faster.
- If there is enough free IRAM, select `CONFIG_LWIP_IRAM_OPTIMIZATION` and `CONFIG_LWIP_EXTRA_IRAM_OPTIMIZATION` to improve TX/RX throughput.

If using a Wi-Fi network interface, please also refer to [Wi-Fi Buffer Usage](#).

#### Minimum Latency

Except for increasing buffer sizes, most changes that increase throughput 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 also causes TCP sockets in the `WAIT_CLOSE` state to be closed and recycled more rapidly when needed to open a new socket, further reducing peak RAM usage.
• Reducing `CONFIG_LWIP_TCP_IP_RECVMBOX_SIZE`, `CONFIG_LWIP_TCP_RECVMBOX_SIZE` and `CONFIG_LWIP_UDP_RECVMBOX_SIZE` reduce RAM usage at the expense of throughput, depending on usage.
• Reducing `CONFIG_LWIP_TCP_MSL` and `CONFIG_LWIP_TCP_FIN_WAIT_TIMEOUT` reduces the maximum segment lifetime in the system. This also causes TCP sockets in the `TIME_WAIT` and `FIN_WAIT_2` states to be closed and recycled more rapidly.
• Disabling `CONFIG_LWIP_IPV6` can save about 39 KB for firmware size and 2 KB RAM when the system is powered up and 7 KB RAM when the TCP/IP stack is running. If there is no requirement for supporting IPv6, it can be disabled to save flash and RAM footprint.
• Disabling `CONFIG_LWIP_IPV4` can save about 26 KB of firmware size and 600 B RAM on power up and 6 KB RAM when the TCP/IP stack is running. If the local network supports IPv6-only configuration, IPv4 can be disabled to save flash and RAM footprint.

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:

\[
lwip_dynamic_peek_memory = (lwip_udp_con_num * lwip_udp_conn) + (lwip_tcp_con_num * (lwip_tcp_tx_win_size + lwip_tcp_rx_win_size + lwip_tcp_conn))
\]

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 occurs (e.g., a sending failure). 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.

### 4.20  Memory Types

ESP32-S2 chip has multiple memory types and flexible memory mapping features. This section describes how ESP-IDF uses these features by default.

ESP-IDF distinguishes between instruction memory bus (IRAM, IROM, RTC FAST memory) and data memory bus (DRAM, DROM). Instruction memory is executable, and can only be read or written via 4-byte aligned words. Data memory is not executable and can be accessed via individual byte operations. For more information about the different memory buses consult the ESP32-S2 Technical Reference Manual > System and Memory [PDF].

#### 4.20.1  DRAM (Data RAM)

Non-constant static data (.data) and zero-initialized data (.bss) is placed by the linker into Internal SRAM as data memory. The remaining space in this region is used for the runtime heap.

By applying the `EXT_RAM_BSS_ATTR` macro, zero-initialized data can also be placed into external RAM. To use this macro, the `CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY` needs to be enabled. See `Allow .bss Segment to Be Placed in External Memory`. 

---

Submit Document Feedback
**Note:** The maximum statically allocated DRAM size is reduced by the IRAM (Instruction RAM) size of the compiled application. The available heap memory at runtime is reduced by the total static IRAM and DRAM usage of the application.

Constant data may also be placed into DRAM, for example if it is used in an non-flash-safe ISR (see explanation under How to Place Code in IRAM).

**“noinit” DRAM**

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 should keep its value after software restart.

Example:

```c
__NOINIT_ATTR uint32_t noinit_data;
```

### 4.20.2 IRAM (Instruction RAM)

ESP-IDF allocates part of the Internal SRAM region for instruction RAM. The region is defined in [*ESP32-S2 Technical Reference Manual > System and Memory > Internal Memory*](#). Except for the first block (up to 32 KB) which is used for MMU cache, the rest of this memory range is used to store parts of application which need to run from RAM.

**Note:** Any internal SRAM which is not used for Instruction RAM will be made available as DRAM (Data RAM) for static data and dynamic allocation (heap).

### When to Place Code in IRAM

Cases when parts of the application should be placed into IRAM:

- Interrupt handlers must be placed into IRAM if `ESP_INTR_FLAG_IRAM` is used when registering the interrupt handler. For more information, see [IRAM-Safe Interrupt Handlers](#).
- Some timing critical code may be placed into IRAM to reduce the penalty associated with loading the code from flash. ESP32-S2 reads code and data from flash via the MMU cache. In some cases, placing a function into IRAM may reduce delays caused by a cache miss and significantly improve that function’s performance.

### How to Place Code in IRAM

Some code is automatically placed into the IRAM region using the linker script.

If some specific application code needs to be placed into IRAM, it can be done by using the [Linker Script Generation](#) feature and adding a linker script fragment file to your component that targets at the entire source files or functions with the `noflash` placement. See the [Linker Script Generation](#) docs for more information.

Alternatively, it is possible to specify IRAM placement in the source code using the `IRAM_ATTR` macro:

```c
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

There are some possible issues with placement in IRAM, that may cause problems with IRAM-safe interrupt handlers:
Strings or constants inside an IRAM_ATTR function may not be placed in RAM automatically. It is possible to use DRAM_ATTR attributes to mark these, or using the linker script method will cause these to be automatically placed correctly.

```c
void IRAM_ATTR gpio_isr_handler(void* arg)
{
    const static DRAM_ATTR uint8_t INDEX_DATA[] = { 45, 33, 12, 0 }; // DRAM_ATTR is used to mark these
    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.

- GCC optimizations that automatically generate jump tables or switch/case lookup tables place these tables in flash. IDF by default builds all files with -fno-jump-tables -fno-tree-switch-conversion flags to avoid this.

Jump table optimizations can be re-enabled for individual source files that do not need to be placed in IRAM. For instructions on how to add the -fno-jump-tables -fno-tree-switch-conversion options when compiling individual source files, see Controlling Component Compilation.

### 4.20.3 IROM (Code Executed from flash)

If a function is not explicitly placed into IRAM (Instruction RAM) or RTC memory, it is placed into flash. As IRAM is limited, most of an application’s binary code must be placed into IROM instead.

During Application Startup Flow, the bootloader (which runs from IRAM) configures the MMU flash cache to map the app’s instruction code region to the instruction space. Flash accessed via the MMU is cached using some internal SRAM and accessing cached flash data is as fast as accessing other types of internal memory.

### 4.20.4 DROM (Data Stored in flash)

By default, constant data is placed by the linker into a region mapped to the MMU flash cache. This is the same as the IROM (Code Executed from flash) section, but is for read-only data not executable code.

The only constant data not placed into this memory type by default are literal constants which are embedded by the compiler into application code. These are placed as the surrounding function’s executable instructions.

The DRAM_ATTR attribute can be used to force constants from DROM into the DRAM (Data RAM) section (see above).

### 4.20.5 RTC Slow Memory

Global and static variables used by code which runs from RTC memory must be placed into RTC Slow memory. For example deep sleep variables can be placed here instead of RTC FAST memory, or code and variables accessed by the ULP Coprocessor Programming.

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;
```
4.20.6 RTC FAST Memory

The same region of RTC FAST memory can be accessed as both instruction and data memory. 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.

Remaining RTC FAST memory is added to the heap unless the option `CONFIG_ESP_SYSTEM_ALLOW_RTC_FAST_MEM_AS_HEAP` is disabled. This memory can be used interchangeably with DRAM (Data RAM), but is slightly slower to access.

4.20.7 DMA-Capable Requirement

Most peripheral 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:

```c
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:

```c
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),
    };
    spi_device_transmit(spi, &temp);
    // other stuff
}
```

It is also possible to allocate DMA-capable memory buffers dynamically by using the `MALLOC_CAP_DMA` capabilities flag.

4.20.8 DMA Buffer in the Stack

Placing DMA buffers in the stack is possible but discouraged. If doing so, pay attention to the following:

- Placing DRAM buffers on the stack is not recommended if the stack may be in 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.
- Use macro `WORD_ALIGNED_ATTR` in functions before variables to place them in proper positions like:

```c
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.
}
```
4.21 OpenThread

OpenThread is an IP stack running on the 802.15.4 MAC layer which features mesh network and low power consumption.

4.21.1 Modes of the OpenThread Stack

OpenThread can run under the following modes on Espressif chips:

Standalone Node

The full OpenThread stack and the application layer run on the same chip. This mode is available on chips with 15.4 radio such as ESP32-H2 and ESP32-C6.

Radio Co-Processor (RCP)

The chip is connected to another host running the OpenThread IP stack. It sends and receives 15.4 packets on behalf of the host. This mode is available on chips with 15.4 radio such as ESP32-H2 and ESP32-C6. The underlying transport between the chip and the host can be SPI or UART. For the sake of latency, we recommend using SPI as the underlying transport.

OpenThread Host

For chips without a 15.4 radio, it can be connected to an RCP and run OpenThread under host mode. This mode enables OpenThread on Wi-Fi chips such as ESP32, ESP32-S2, ESP32-S3, and ESP32-C3. The following diagram shows how devices work under different modes:

4.21.2 How to Write an OpenThread Application

The OpenThread `openthread/ot_cli` example is a good place to start at. It demonstrates basic OpenThread initialization and simple socket-based server and client.

Before OpenThread Initialization

- s1.1: The main task calls `esp_vfs_eventfd_register()` to initialize the eventfd virtual file system. The eventfd file system is used for task notification in the OpenThread driver.
- s1.2: The main task calls `nvs_flash_init()` to initialize the NVS where the Thread network data is stored.
- s1.3: Optional. The main task calls `esp_netif_init()` only when it wants to create the network interface for Thread.
• s1.4: The main task calls `esp_event_loop_create()` to create the system Event task and initialize an application event’s callback function.

OpenThread Stack Initialization

• s2.1: Call `esp_openthread_init()` to initialize the OpenThread stack.

OpenThread Network Interface Initialization

The whole stage is optional and only required if the application wants to create the network interface for Thread.

• s3.1: Call `esp_netif_new()` with `ESP_NETIF_DEFAULT_OPENTHREAD` to create the interface.
• s3.2: Call `esp_openthread_netif_glue_init()` to create the OpenThread interface handlers.
• s3.3: Call `esp_netif_attach()` to attach the handlers to the interface.

The OpenThread Main Loop

• s4.3: Call `esp_openthread_launch_mainloop()` to launch the OpenThread main loop. Note that this is a busy loop and does not return until the OpenThread stack is terminated.

Calling OpenThread APIs

The OpenThread APIs are not thread-safe. When calling OpenThread APIs from other tasks, make sure to hold the lock with `esp_openthread_lock_acquire()` and release the lock with `esp_openthread_lock_release()` afterwards.

Deinitialization

The following steps are required to deinitialize the OpenThread stack:

• Call `esp_netif_destroy()` and `esp_openthread_netif_glue_deinit()` to deinitialize the OpenThread network interface if you have created one.
• Call `esp_openthread_deinit()` to deinitialize the OpenThread stack.
4.21.3 The OpenThread Border Router

The OpenThread border router connects the Thread network with other IP networks. It provides IPv6 connectivity, service registration, and commission functionality.

To launch an OpenThread border router on an ESP chip, you need to connect an RCP to a Wi-Fi capable chip such as ESP32.

Calling `esp_openthread_border_router_init()` during the initialization launches all the border routing functionalities.

You may refer to the `openthread/ot_br` example and the README for further border router details.

4.22 Partition Tables

4.22.1 Overview

A single ESP32-S2’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.

The partition table length is 0xC00 bytes, as we allow a maximum of 95 entries. An MD5 checksum, used for checking the integrity of the partition table at runtime, is appended after the table data. Thus, the partition table occupies an entire flash sector, which size is 0x1000 (4 KB). As a result, any partition following it must be at least located at (default offset) + 0x1000.

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.

4.22.2 Built-in Partition Tables

Here is the summary printed for the “Single factory app, no OTA” configuration:

```
# ESP-IDF 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 (64 KB) 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:

```
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 0x4000,
```

(continues on next page)
Chapter 4. API Guides

(continued from previous page)

otadata, data, ota, 0xd000, 0x2000,
phy_init, data, phy, 0xf000, 0x1000,
factory, app, factory, 0x10000, 1M,
ota_0, app, ota_0, 0x10000, 1M,
ota_1, app, ota_1, 0x20000, 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.

4.22.3 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:

<table>
<thead>
<tr>
<th>Name, Type, SubType, Offset, Size, Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvs, data, nvs, 0x9000, 0x4000</td>
</tr>
<tr>
<td>otadata, data, ota, 0xd000, 0x2000</td>
</tr>
<tr>
<td>phy_init, data, phy, 0xf000, 0x1000</td>
</tr>
<tr>
<td>factory, app, factory, 0x10000, 1M</td>
</tr>
<tr>
<td>ota_0, app, ota_0, , 1M</td>
</tr>
<tr>
<td>ota_1, app, ota_1, , 1M</td>
</tr>
<tr>
<td>nvs_key, data, nvs_keys, , 0x1000</td>
</tr>
</tbody>
</table>

• 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-S2. The maximum length of names is 16 bytes, including one null terminator. Names longer than the maximum length will be truncated.

Type Field

Partition type field can be specified as app (0x00) or data (0x01). 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 app needs to store data in a format not already supported by ESP-IDF, then please add a custom partition type value in the range 0x40-0xFE.

See esp_partition_type_t for the enum definitions for app and data partitions.

If writing in C++ then specifying a application-defined partition type requires casting an integer to esp_partition_type_t in order to use it with the partition API. For example:

static const esp_partition_type_t APP_PARTITION_TYPE_A = (esp_partition_type_t)0x40;

The ESP-IDF bootloader ignores any partition types other than app (0x00) and data (0x01).
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.

See enum `esp_partition_subtype_t` for the full list of subtypes defined by ESP-IDF, including the following:

- When type is app, the SubType field can be specified as `factory` (0x00), `ota_0` (0x10) … `ota_15` (0x1F) or `test` (0x20).
  - `factory` (0x00) 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. When OTA is in use, the OTA data partition configures which app slot the bootloader should boot. When 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` (0x00), `phy` (0x01), `nvs` (0x02), `nvs_keys` (0x04), or a range of other component-specific subtypes (see subtype enum).
  - `ota` (0) is the OTA data partition which stores information about the currently selected OTA app slot. 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_ESP_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 Wi-Fi data if the `esp_wifi_set_storage(WIFI_STORAGE_FLASH)` initialization 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).
- There are other predefined data subtypes for data storage supported by ESP-IDF. These include:
  - `coredump` (0x03) is for storing core dumps while using a custom partition table CSV file. See Core Dump for more details.
  - `efuse` (0x05) is for emulating eFuse bits using Virtual eFuses.
  - `undefined` (0x06) is implicitly used for data partitions with unspecified (empty) subtype, but it is possible to explicitly mark them as undefined as well.
  - `fat` (0x81) is for FAT Filesystem Support.
  - `spiffs` (0x82) is for SPIFFS Filesystem.
  - `littlefs` (0x83) is for LittleFS filesystem. See storage/littlefs example for more details.
- If the partition type is any application-defined value (range 0x40-0xFE), then subtype field can be any value chosen by the application (range 0x00-0xFE).

Note that when writing in C++, an application-defined subtype value requires casting to type `esp_partition_subtype_t` in order to use it with the partition API.
Extra Partition SubTypes

A component can define a new partition subtype by setting the `EXTRA_PARTITION_SUBTYPES` property. This property is a CMake list, each entry of which is a comma separated string with `<type>`, `<subtype>`, `<value>` format. The build system uses this property to add extra subtypes and creates fields named `ESP_PARTITION_SUBTYPE_<type>_<subtype>` in `esp_partition_subtype_t`. The project can use this subtype to define partitions in the partitions table CSV file and use the new fields in `esp_partition_subtype_t`.

Offset & Size

The offset represents the partition address in the SPI flash, which sector size is 0x1000 (4 KB). Thus, the offset must be a multiple of 4 KB.

Partitions with blank offsets in the CSV file will start after the previous partition, or after the partition table in the case of the first partition.

Partitions of type `app` have to be placed at offsets aligned to 0x10000 (64 K). 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 relative to any placement (`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

Two flags are currently supported, `encrypted` and `readonly`:

- If `encrypted` flag is set, the partition will be encrypted if Flash Encryption is enabled.

| Note: app type partitions will always be encrypted, regardless of whether this flag is set or not. |

- If `readonly` flag is set, the partition will be read-only. This flag is only supported for `data` type partitions except `ota` and `coredump` subtypes. This flag can help to protect against accidental writes to a partition that contains critical device-specific configuration data, e.g., factory data partition.

| Note: Using C file I/O API to open a file (`fopen`) in any write mode (w, w+, a, a+, r+) will fail and return NULL. Using open with any other flag than 0_RDONLY will fail and return -1 while errno global variable will be set to EROFS. This is also true for any other POSIX syscall function performing write or erase operations. Opening a handle in read-write mode for NVS on a read-only partition will fail and return ESP_ERR_NOT_ALLOWED error code. Using a lower level API like esp_partition, spi_flash, etc. to write to a read-only partition will result in ESP_ERR_NOT_ALLOWED error code. |

You can specify multiple flags by separating them with a colon. For example, `encrypted:readonly`.

4.22.4 Generating Binary Partition Table

The partition table which is flashed to the ESP32-S2 is in a binary format, not CSV. The tool `partition_table/gen.esp32part.py` is used to convert between CSV and binary formats.
Chapter 4. API Guides

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
```

### 4.22.5 Partition Size Checks

The ESP-IDF build system will automatically check if generated binaries fit in the available partition space, and will fail with an error if a binary is too large.

Currently these checks are performed for the following binaries:

- Bootloader binary must fit in space before partition table (see Bootstrap Size).
- App binary should fit in at least one partition of type “app”. If the app binary does not fit in any app partition, the build will fail. If it only fits in some of the app partitions, a warning is printed about this.

**Note:** Although the build process will fail if the size check returns an error, the binary files are still generated and can be flashed (although they may not work if they are too large for the available space.)

#### 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.

### 4.22.6 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:** Note that updating the partition table does not 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.

### 4.22.7 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 name, offset, size and flag ("encrypted") 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.

```python
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:

```python
# 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:

```python
# 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 `parttool.py` has the following structure:

```
parttool.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
Chapter 4. API Guides

```bash
# Erase partition with name 'storage'
parttool.py --port "/dev/ttyUSB1" erase_partition --partition-name=storage

# Read partition with type 'data' and subtype 'spiffs' and save to file 'spiffs.bin'
parttool.py --port "/dev/ttyUSB1" read_partition --partition-type=data --partition=subtype=spiffs --output "spiffs.bin"

# Write to partition 'factory' the contents of a file named 'factory.bin'
parttool.py --port "/dev/ttyUSB1" write_partition --partition-name=factory --input="factory.bin"

# Print the size of default boot partition
parttool.py --port "/dev/ttyUSB1" get_partition_info --partition-boot-default --info size
```

More information can be obtained by specifying `--help` as argument:

```bash
# Display possible subcommands and show main command argument descriptions
parttool.py --help

# Show descriptions for specific subcommand arguments
parttool.py [subcommand] --help
```

### 4.23 Performance

ESP-IDF ships with default settings that are designed for a trade-off between performance, resource usage, and available functionality.

These guides describe how to optimize a firmware application for a particular aspect of performance. Usually this involves some trade-off in terms of limiting available functions, or swapping one aspect of performance (such as execution speed) for another (such as RAM usage).

#### 4.23.1 How to Optimize Performance

1. Decide the performance-critical aspects of your application, such as achieving a particular response time for a certain network operation, meeting a particular startup time limit, or maintaining a certain level of peripheral data throughput.
2. Find a way to measure this performance (some methods are outlined in the guides below).
3. Modify the code and project configuration and compare the new measurement to the old measurement.
4. Repeat step 3 until the performance meets the requirements set out in step 1.

#### 4.23.2 Guides

**Speed Optimization**

**Overview** Optimizing execution speed is a key element of software performance. Code that executes faster can also have other positive effects, e.g., reducing overall power consumption. However, improving execution speed may have trade-offs with other aspects of performance such as *Minimizing Binary Size*. 
Choose What to Optimize  If a function in the application firmware is executed once per week in the background, it may not matter if that function takes 10 ms or 100 ms to execute. If a function is executed constantly at 10 Hz, it matters greatly if it takes 10 ms or 100 ms to execute.

Most kinds of application firmware only have a small set of functions that require optimal performance. Perhaps those functions are executed very often, or have to meet some application requirements for latency or throughput. Optimization efforts should be targeted at these particular functions.

Measuring Performance  The first step to improving something is to measure it.

Basic Performance Measurements  You may be able to measure directly the performance relative to an external interaction with the world, e.g., see the examples wifi/iperf and ethernet/iperf for measuring general network performance. Or you can use an oscilloscope or logic analyzer to measure the timing of an interaction with a device peripheral.

Otherwise, one way to measure performance is to augment the code to take timing measurements:

```c
#include "esp_timer.h"

void measure_important_function(void) {
    const unsigned MEASUREMENTS = 5000;
    uint64_t start = esp_timer_get_time();

    for (int retries = 0; retries < MEASUREMENTS; retries++) {
        important_function(); // This is the thing you need to measure
    }

    uint64_t end = esp_timer_get_time();

    printf("%u iterations took %llu milliseconds (%llu microseconds per...
            --invocation)\n", MEASUREMENTS, (end - start)/1000, (end - start)/MEASUREMENTS);
}
```

Executing the target multiple times can help average out factors, e.g., RTOS context switches, overhead of measurements, etc.

- Using `esp_timer_get_time()` generates “wall clock” timestamps with microsecond precision, but has moderate overhead each time the timing functions are called.
- It is also possible to use the standard Unix `gettimeofday()` and `utime()` functions, although the overhead is slightly higher.
- Otherwise, including `hal/cpu_hal.h` and calling the HAL function `cpu_hal_get_cycle_count()` returns the number of CPU cycles executed. This function has lower overhead than the others, which is good for measuring very short execution times with high precision.
- While performing “microbenchmarks” (i.e., benchmarking only a very small routine of code that runs in less than 1-2 milliseconds), the flash cache performance can sometimes cause big variations in timing measurements depending on the binary. This happens because binary layout can cause different patterns of cache misses in a particular sequence of execution. If the test code is larger, then this effect usually averages out. Executing a small function multiple times when benchmarking can help reduce the impact of flash cache misses. Alternatively, move this code to IRAM (see Targeted Optimizations).

External Tracing  The Application Level Tracing Library allows measuring code execution with minimal impact on the code itself.

Tasks  If the option `CONFIG_FREERTOS_GENERATE_RUN_TIME_STATS` is enabled, then the FreeRTOS API `vTaskGetRunTimeStats()` can be used to retrieve runtime information about the processor time used by each FreeRTOS task.
**Segger SystemView** is an excellent tool for visualizing task execution and looking for performance issues or improvements in the system as a whole.

**Improving Overall Speed**  The following optimizations improve the execution of nearly all code, including boot times, throughput, latency, etc:

- Set `CONFIG_ESPTOOLPY_FLASHMODE` to QIO or QOUT mode (Quad I/O). Both almost double the speed at which code is loaded or executed from flash compared to the default DIO mode. QIO is slightly faster than QOUT if both are supported. Note that both the flash chip model, and the electrical connections between the ESP32-S2 and the flash chip must support quad I/O modes or the SoC will not work correctly.

- Set `CONFIG_COMPILER_OPTIMIZATION` to `Optimize for performance (-O2)`. This may slightly increase binary size compared to the default setting, but almost certainly increases the performance of some code. Note that if your code contains C or C++ Undefined Behavior, then increasing the compiler optimization level may expose bugs that otherwise are not seen.

- Avoid using floating point arithmetic `float`. On ESP32-S2 these calculations are emulated in software and are very slow. If possible, use fixed point representations, a different method of integer representation, or convert part of the calculation to be integer only before switching to floating point.

- Avoid using double precision floating point arithmetic `double`. These calculations are emulated in software and are very slow. If possible then use an integer-based representation, or single-precision floating point.

**Reduce Logging Overhead**  Although standard output is buffered, it is possible for an application to be limited by the rate at which it can print data to log output once buffers are full. This is particularly relevant for startup time if a lot of output is logged, but such problem can happen at other times as well. There are multiple ways to solve this problem:

- Reduce the volume of log output by lowering the app `CONFIG_LOG_DEFAULT_LEVEL` (the equivalent bootloader setting is `CONFIG_BOOTLOADER_LOG_LEVEL`). This also reduces the binary size, and saves some CPU time spent on string formatting.

- Increase the speed of logging output by increasing the `CONFIG_ESP_CONSOLE_UART_BAUDRATE`. However, if you are using internal USB-CDC, the serial throughput is not dependent on the configured baud rate.

**Not Recommended**  The following options also increase execution speed, but are not recommended as they also reduce the debuggability of the firmware application and may increase the severity of any bugs.

- Set `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL` to disabled. This also reduces firmware binary size by a small amount. However, it may increase the severity of bugs in the firmware including security-related bugs. If it is necessary to do this to optimize a particular function, consider adding `#define NDEBUG` at the top of that single source file instead.

**Targeted Optimizations**  The following changes increase the speed of a chosen part of the firmware application:

- Move frequently executed code to IRAM. By default, all code in the app is executed from flash cache. This means that it is possible for the CPU to have to wait on a “cache miss” while the next instructions are loaded from flash. Functions which are copied into IRAM are loaded once at boot time, and then always execute at full speed.

  IRAM is a limited resource, and using more IRAM may reduce available DRAM, so a strategic approach is needed when moving code to IRAM. See **IRAM (Instruction RAM)** for more information.

- Jump table optimizations can be re-enabled for individual source files that do not need to be placed in IRAM. For hot paths in large switch cases, this improves performance. For instructions on how to add the `-fjump-tables` and `-ftree-switch-conversion` options when compiling individual source files, see **Controlling Component Compilation**.
Improving Startup Time  In addition to the overall performance improvements shown above, the following options can be tweaked to specifically reduce startup time:

- Minimizing the `CONFIG_LOG_DEFAULT_LEVEL` and `CONFIG_BOOTLOADER_LOG_LEVEL` has a large impact on startup time. To enable more logging after the app starts up, set the `CONFIG_LOG_MAXIMUM_LEVEL` as well, and then call `esp_log_level_set()` to restore higher level logs. The `system/startup_time` main function shows how to do this.
- If using Deep-sleep mode, setting `CONFIG_BOOTLOADER_SKIP_VALIDATE_IN_DEEP_SLEEP` allows a faster wake from sleep. Note that if using Secure Boot, this represents a security compromise, as Secure Boot validation are not be performed on wake.
- Setting `CONFIG_BOOTLOADER_SKIP_VALIDATE_ON_POWER_ON` skips verifying the binary on every boot from the power-on reset. How much time this saves depends on the binary size and the flash settings. Note that this setting carries some risk if the flash becomes corrupt unexpectedly. Read the help text of the config item for an explanation and recommendations if using this option.
- It is possible to save a small amount of time during boot by disabling RTC slow clock calibration. To do so, set `CONFIG_RTC_CLK_CAL_CYCLES` to 0. Any part of the firmware that uses RTC slow clock as a timing source will be less accurate as a result.

The example project `system/startup_time` is pre-configured to optimize startup time. The file `system/startup_time/sdkconfig.defaults` contain all of these settings. You can append these to the end of your project’s own `sdkconfig` file to merge the settings, but please read the documentation for each setting first.

Task Priorities  As ESP-IDF FreeRTOS is a real-time operating system, it is necessary to ensure that high-throughput or low-latency tasks are granted a high priority in order to run immediately. Priority is set when calling `xTaskCreate()` or `xTaskCreatePinnedToCore()` and can be changed at runtime by calling `vTaskPrioritySet()`.

It is also necessary to ensure that tasks yield CPU (by calling `vTaskDelay()`, `sleep()`, or by blocking on semaphores, queues, task notifications, etc) in order to not starve lower-priority tasks and cause problems for the overall system. The Task Watchdog Timer (TWDT) provides a mechanism to automatically detect if task starvation happens. However, note that a TWDT timeout does not always indicate a problem, because sometimes the correct operation of the firmware requires some long-running computation. In these cases, tweaking the TWDT timeout or even disabling the TWDT may be necessary.

Built-in Task Priorities  ESP-IDF starts a number of system tasks at fixed priority levels. Some are automatically started during the boot process, while some are started only if the application firmware initializes a particular feature. To optimize performance, structure the task priorities of your application properly to ensure the tasks are not delayed by the system tasks, while also not starving system tasks and impacting other functions of the system.

This may require splitting up a particular task. For example, perform a time-critical operation in a high-priority task or an interrupt handler and do the non-time-critical part in a lower-priority task.

Header `components/esp_system/include/esp_task.h` contains macros for the priority levels used for built-in ESP-IDF tasks system. See Background Tasks for more details about the system tasks.

Common priorities are:

- **Running the Main Task** that executes app_main function has minimum priority (1).
- **High Resolution Timer (ESP Timer)** system task to manage timer events and execute callbacks has high priority (22, `ESP_TASK_TIMER_PRIO`)
- FreeRTOS Timer Task to handle FreeRTOS timer callbacks is created when the scheduler initializes and has minimum task priority (1, configurable).
- **Event Loop Library** system task to manage the default system event loop and execute callbacks has high priority (20, `ESP_TASK_EVENT_PRIO`). This configuration is only used if the application calls `esp_event_loop_create_default()`. It is possible to call `esp_event_loop_create()` with a custom task configuration instead.
- **lwIP** TCP/IP task has high priority (18, `ESP_TASK_TCPIP_PRIO`).
• **Wi-Fi Driver** task has high priority (23).
• Wi-Fi wpa_supplicant component may create dedicated tasks while the Wi-Fi Protected Setup (WPS), WPA2 EAP-TLS, Device Provisioning Protocol (DPP) or BSS Transition Management (BTM) features are in use. These tasks all have low priority (2).
• The Ethernet driver creates a task for the MAC to receive Ethernet frames. If using the default config ETH_MAC_DEFAULT_CONFIG then the priority is medium-high (15). This setting can be changed by passing a custom eth_mac_config_t struct when initializing the Ethernet MAC.
• If using the ESP-MQTT component, it creates a task with default priority 5 (configurable), depending on CONFIG_MQTT_USE_CUSTOM_CONFIG, and also configurable at runtime by task_prio field in the esp_mqtt_client_config_t
• To see what is the task priority for mDNS service, please check Performance Optimization.

**Choosing Task Priorities of the Application**  
In general, it is not recommended to set task priorities higher than the built-in Wi-Fi operations as starving them of CPU may make the system unstable. For very short timing-critical operations that do not use the network, use an ISR or a very restricted task (with very short bursts of runtime only) at the highest priority (24). Choosing priority 19 allows lower-layer Wi-Fi functionality to run without delays, but still preempts the lwIP TCP/IP stack and other less time-critical internal functionality - this is the best option for time-critical tasks that do not perform network operations. Any task that does TCP/IP network operations should run at a lower priority than the lwIP TCP/IP task (18) to avoid priority-inversion issues.

**Note:** Task execution is always completely suspended when writing to the built-in SPI flash chip. Only IRAM-Safe Interrupt Handlers continues executing.

**Improving Interrupt Performance**  
ESP-IDF supports dynamic Interrupt Allocation with interrupt preemption. Each interrupt in the system has a priority, and higher-priority interrupts preempts lower priority ones.

Interrupt handlers execute in preference to any task, provided the task is not inside a critical section. For this reason, it is important to minimize the amount of time spent in executing an interrupt handler.

To obtain the best performance for a particular interrupt handler:

• Assign more important interrupts a higher priority using a flag such as ESP_INTR_FLAG_LEVEL2 or ESP_INTR_FLAG_LEVEL3 when calling esp_intr_alloc().
• If you are sure the entire interrupt handler can run from IRAM (see IRAM-Safe Interrupt Handlers) then set the ESP_INTR_FLAG_IRAM flag when calling esp_intr_alloc() to assign the interrupt. This prevents it being temporarily disabled if the application firmware writes to the internal SPI flash.
• Even if the interrupt handler is not IRAM-safe, if it is going to be executed frequently then consider moving the handler function to IRAM anyhow. This minimizes the chance of a flash cache miss when the interrupt code is executed (see Targeted Optimizations). It is possible to do this without adding the ESP_INTR_FLAG_IRAM flag to mark the interrupt as IRAM-safe, if only part of the handler is guaranteed to be in IRAM.

**Improving Network Speed**

• For Wi-Fi, see How to Improve Wi-Fi Performance and Wi-Fi Buffer Usage
• For lwIP TCP/IP (Wi-Fi and Ethernet), see Performance Optimization
• The wifi/iperf example contains a configuration that is heavily optimized for Wi-Fi TCP/IP throughput. Append the contents of the files wifi/iperf/sdkconfig.defaults, wifi/iperf/sdkconfig.defaults.esp32s2 and wifi/iperf/sdkconfig.ci.99 to the sdkconfig file in your project in order to add all of these options. Note that some of these options may have trade-offs in terms of reduced debuggability, increased firmware size, increased memory usage, or reduced performance of other features. To get the best result, read the documentation pages linked above and use related information to determine exactly which options are best suited for your app.
Improving I/O Performance  Using standard C library functions like `fread` and `fwrite` instead of platform specific unbuffered syscalls such as `read` and `write` can be slow. These functions are designed to be portable, so they are not necessarily optimized for speed, have a certain overhead and are buffered.

FAT Filesystem Support  specific information and tips:

- Maximum size of the R/W request = FatFS cluster size (allocation unit size).
- Use `read` and `write` instead of `fread` and `fwrite`.
- To increase speed of buffered reading functions like `fread` and `fgets`, you can increase a size of the file buffer (Newlib’s default is 128 bytes) to a higher number like 4096, 8192 or 16384. This can be done locally via the `setvbuf` function used on a certain file pointer or globally applied to all files via modifying `CONFIG_FATFS_VFS_FSTAT_BLKSIZE`.

Note: Setting a bigger buffer size also increases the heap memory usage.

Minimizing Binary Size

The ESP-IDF build system compiles all source files in the project and ESP-IDF, but only functions and variables that are actually referenced by the program are linked into the final binary. In some cases, it is necessary to reduce the total size of the firmware binary, e.g., in order to fit it into the available flash partition size.

The first step to reducing the total firmware binary size is measuring what is causing the size to increase.

Measuring Static Sizes  To optimize both the firmware binary size and the memory usage, it is necessary to measure statically-allocated RAM (data, bss), code (text), and read-only data (rodata) in your project.

Using the `idf.py` sub-commands `size`, `size-components`, and `size-files` provides a summary of memory used by the project:

Note: It is possible to add `-DOUTPUT_FORMAT=csv` or `-DOUTPUT_FORMAT=json` to get the output in CSV or JSON format.

Size Summary `idf.py size`

```
$ idf.py size
[...]  
Total sizes:
Used stat D/IRAM:  53743 bytes ( 122385 remain, 30.5% used)  
    .data size:  6504 bytes
    .bss size:  1984 bytes
    .text size:  44228 bytes
    .vectors size:  1027 bytes
Used Flash size:  118879 bytes  
    .text:  83467 bytes
    .rodata:  35156 bytes
Total image size:  170638 bytes (.bin may be padded larger)
```

This output breaks down the size of all static memory regions in the firmware binary:

```
$ idf.py size
[...]  
Total sizes:
Used stat D/IRAM:  53743 bytes ( 122385 remain, 30.5% used)  
    .data size:  6504 bytes
```
Chapter 4. API Guides

(continued from previous page)

<table>
<thead>
<tr>
<th>Component</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bss</td>
<td>1984 bytes</td>
</tr>
<tr>
<td>.text</td>
<td>44228 bytes</td>
</tr>
<tr>
<td>.vectors</td>
<td>1027 bytes</td>
</tr>
<tr>
<td>Used Flash size</td>
<td>11879 bytes</td>
</tr>
<tr>
<td>.text</td>
<td>83467 bytes</td>
</tr>
<tr>
<td>.rodata</td>
<td>35156 bytes</td>
</tr>
<tr>
<td>Total image size</td>
<td>170638 bytes</td>
</tr>
</tbody>
</table>

- **Used stat D/IRAM**: Total amount of D/IRAM used at compile time. *remain* indicates the amount of D/IRAM left to be used as heap memory at runtime. Note that due to meta data overhead, implementation constraints, and startup heap allocations, the actual size of the DRAM heap is smaller.
  - **.data size**: Amount of D/IRAM allocated at compile time for the *data* (i.e., all statically allocated variables that are initialized to non-zero values). *data* also consumes space in the binary image to store the non-zero initialization values.
  - **.bss size**: Amount of D/IRAM allocated at compile time for *bss* (i.e., all statically allocated variables that are initialized to zero). *bss* does not consume extra space in flash.
  - **.text size**: Amount of D/IRAM used for *text* (i.e., all code that is executed from internal RAM). *text* also consumes space in the binary image as the code is initially stored there and is then copied over to D/IRAM on startup.

- **Used Flash size**: Total amount of flash used (excluding usage by D/IRAM)
  - **.text**: Amount of flash used for *text* (i.e., all code that is executed via the flash cache, see *IROM*).
  - **.rodata**: Amount of flash used for *rodata* (i.e., read-only data that is loaded via the flash cache, see *DROM*).

- **Total image size** is the estimated total size of the binary file.

**Component Usage Summary idf.py size-components** The summary output provided by *idf.py size* does not give enough details to find the main contributor to excessive binary size. To analyze in detail, use *idf.py size-components*.

```
$ idf.py size-components

[...]

Total sizes:
DRAM .data size: 14956 bytes
DRAM .bss size: 15808 bytes
Used static DRAM: 30764 bytes (149972 available, 17.0% used)
Used static IRAM: 83918 bytes (47154 available, 64.0% used)
Flash code: 559943 bytes
Flash rodata: 176736 bytes
Total image size: 835553 bytes (.bin may be padded larger)
```

---

Per-archive contributions to ELF file:

```
   Archive File       DRAM .data & .bss & other | IRAM | D/IRAM | Flash code &...
  -------------------+-------------------------------------------+-----+--------+
         Total        1267 6044 0 5490 0 107445 0
     libnet80211.a   21 3838 0 0 0 97465 0
     liblwip.a        60 524 0 0 0 27655 0
     libmbedtls.a     64 81 0 30 0 76645 0
     libmbedtlscrypto.a 12 848 0 0 0 35505 0
     libpp.a           4 0 0 0 0 57056 0
     libc.a            1439 715 0 7798 0 33074 0
     libphy.a          12 848 0 0 0 35505 0
     libwpa_supplicant.a 3104 740 0 15711 0 367 0
```
The first lines of the output from `idf.py size-components` are the same as that from `idf.py size`. After this, a table is printed as `Per-archive contributions to ELF file`. This means how much each static library archive has contributed to the final binary size.

Generally, one static library archive is built per component, although some are binary libraries included by a particular component, for example, `libnet80211.a` is included by `esp_wifi` component. There are also toolchain libraries such as `libc.a` and `libgcc.a` listed here, these provide Standard C/C++ Library and toolchain built-in functionality.

If your project is simple and only has a `main` component, then all of the project’s code will be shown under `libmain.a`. If your project includes its own components (see `Build System`), then they will each be shown on a separate line.

The table is sorted in descending order of the total contribution of the static archive to the binary size.

The columns are as follows:

- **DRAM .data & .bss & other**: .data and .bss are the same as for the totals shown above. Both are static variables and reduce the total available RAM at runtime, but .bss does not contribute to the binary file size. other is a column for any custom section types that also contribute to RAM size. Usually, the value is 0.
- **IRAM**: is the same as for the totals shown above. It refers to code linked to execute from IRAM, which uses space in the binary file and also reduces DRAM available as heap at runtime.
- **Flash code & rodata**: these are the same as the totals above, IROM and DROM space accessed from the flash cache that contribute to the binary size.

**Source File Usage Summary `idf.py size-files`** For even more details, run `idf.py size-files` to get a summary of the contribution each object file has made to the final binary size. Each object file corresponds to a single source file.

```
$ idf.py size-files
[...]
Total sizes:
  DRAM .data size: 14956 bytes
  DRAM .bss size: 15088 bytes
  Used static DRAM: 30764 bytes (149972 available, 17.0% used)
  Used static IRAM: 83918 bytes (47154 available, 64.0% used)
```

(continues on next page)
<table>
<thead>
<tr>
<th>File</th>
<th>Total</th>
<th>x509_crt_bundle.S.o</th>
<th>rodata</th>
<th>.data &amp; .bss &amp; other</th>
<th>IRAM</th>
<th>D/IRAM</th>
<th>Flash code &amp; code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash code:</td>
<td>559943</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash rodata:</td>
<td>176736</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total image size:</td>
<td>835553</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(.bin may be padded</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>larger)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per-file contributions to ELF file:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rodata:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code:</td>
<td>15625</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.text:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.data:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.bss:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D/IRAM:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(continues on next page)
After the summary of total sizes, a table of Per-file contributions to ELF file is printed.

The columns are the same as shown above for idy.py size-components, but this time the granularity is the contribution of each individual object file to the binary size.

For example, we can see that the file x509.crt_bundle.S.o contributed 64,212 bytes to the total firmware size, all as .rodata in flash. Therefore we can guess that this application is using the ESP x509 Certificate Bundle feature and not using this feature would save at least this many bytes from the firmware size.

Some of the object files are linked from binary libraries and therefore you will not find a corresponding source file. To locate which component a source file belongs to, it is generally possible to search in the ESP-IDF source tree or look in the Linker Map File for the full path.

Comparing Two Binaries If making some changes that affect binary size, it is possible to use an ESP-IDF tool to break down the exact differences in size.

This operation is not part of idf.py, it is necessary to run the esp_idf_size Python tool directly.

To do so, first, locate the linker map file with the name PROJECTNAME.map in the build directory. The esp_idf_size tool performs its analysis based on the output of the linker map file.

To compare with another binary, you also need its corresponding .map file saved from the build directory.

For example, to compare two builds, one of which with the default CONFIG_COMPILER_OPTIMIZATION setting Debug (–Og) configuration while another with Optimize for size (–Os):

```bash
$ python -m esp_idf_size --diff build_Og/https_request.map build_Os/https_request.map
```

We can see from the Difference column that changing this one setting caused the whole binary to be over 60 KB smaller and over 5 KB more RAM is available.

It is also possible to use the diff mode to output a table of component-level (static library archive) differences:
Note: To get the output in JSON or CSV format using `esp_idf_size`, it is possible to use the `--format` option.

```
python -m esp_idf_size --archives --diff build_Og/https_request.map build_Oshttps_request.map
```

Also at the individual source file level:

```
python -m esp_idf_size --files --diff build_Og/https_request.map build_Oshttps_request.map
```

Other options, like writing the output to a file, are available, pass `--help` to see the full list.

**Showing Size When Linker Fails** If too much static memory is allocated, the linker will fail with an error such as `DRAM segment data does not fit, region `iram0_0_seg' overflowed by 44 bytes`, or similar.

In these cases, `idf.py size` will not succeed either. However, it is possible to run `esp_idf_size` manually to view the partial static memory usage. The memory usage will miss the variables that could not be linked, so there still appears to be some free space.

The map file argument is `<projectname>.map` in the build directory.

```
python -m esp_idf_size build/project_name.map
```

It is also possible to view the equivalent of `size-components` or `size-files` output:

```
python -m esp_idf_size --archives build/project_name.map
python -m esp_idf_size --files build/project_name.map
```

**Linker Map File**

Note: This is an advanced analysis method, but it can be very useful. Feel free to skip ahead to *Reducing Overall Size* and possibly come back to this later.

The `idf.py size` analysis tools all work by parsing the GNU binutils linker map file, which is a summary of everything the linker did when it created (i.e., linked) the final firmware binary file.

Linker map files themselves are plain text files, so it is possible to read them and find out exactly what the linker did. However, they are also very complex and long, often exceeding 100,000 lines.

The map file itself is broken into parts and each part has a heading. The parts are:

- **Archive member included to satisfy reference by file (symbol)**
  - This shows you: for each object file included in the link, what symbol (function or variable) was the linker searching for when it included that object file.
  - If you are wondering why some object file in particular was included in the binary, this part may give a clue. This part can be used in conjunction with the Cross Reference Table at the end of the file.

  Note: Not every object file shown in this list ends up included in the final binary, some end up in the Discarded input sections list instead.

- **Allocating common symbols**
  - This is a list of some global variables along with their sizes. Common symbols have a particular meaning in ELF binary files, but ESP-IDF does not make much use of them.

- **Discarded input sections**
  - These sections were read by the linker as part of an object file to be linked into the final binary, but then nothing else referred to them, so they were discarded from the final binary.
For ESP-IDF, this list can be very long, as we compile each function and static variable to a unique section in order to minimize the final binary size. Specifically, ESP-IDF uses compiler options `-ffunction-sections` `-fdata-sections` and linker option `--gc-sections`.

Items mentioned in this list do not contribute to the final binary.

- **Memory Configuration, Linker script and memory map**
  - These two parts go together. Some of the output comes directly from the linker command line and the Linker Script, both provided by `Build System`. The linker script is partially generated from the ESP-IDF project using the Linker Script Generation feature.
  - As the output of the Linker script and memory map part of the map unfolds, you can see each symbol (function or static variable) linked into the final binary along with its address (as a 16 digit hex number), its length (also in hex), and the library and object file it was linked from (which can be used to determine the component and the source file).
  - Following all of the output sections that take up space in the final `.bin` file, the memory map also includes some sections in the ELF file that are only used for debugging, e.g., ELF sections `.debug_*`, etc. These do not contribute to the final binary size. You can notice the address of these symbols is a very small number, starting from `0x0000000000000000` and counting up.

- **Cross Reference Table**
  - This table shows the symbol (function or static variable) that the list of object file(s) refers to. If you are wondering why a particular thing is included in the binary, this will help determine what included it.

**Note:** Unfortunately, the Cross Reference Table does not only include symbols that made it into the final binary. It also includes symbols in discarded sections. Therefore, just because something is shown here does not mean that it was included in the final binary - this needs to be checked separately.

**Note:** Linker map files are generated by the GNU binutils linker `ld`, not ESP-IDF. You can find additional information online about the linker map file format. This quick summary is written from the perspective of ESP-IDF build system in particular.

### Reducing Overall Size

The following configuration options reduces the final binary size of almost any ESP-IDF project:

- Set `CONFIG_COMPILER_OPTIMIZATION` to Optimize for size (`-Os`). In some cases, Optimize for performance (`-O2`) will also reduce the binary size compared to the default. Note that if your code contains C or C++ Undefined Behavior then increasing the compiler optimization level may expose bugs that otherwise do not happen.
- Reduce the compiled-in log output by lowering the app `CONFIG_LOG_DEFAULT_LEVEL`. If the `CONFIG_LOG_MAXIMUM_LEVEL` is changed from the default then this setting controls the binary size instead. Reducing compiled-in logging reduces the number of strings in the binary, and also the code size of the calls to logging functions.
- Set the `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL` to Silent. This avoids compiling in a dedicated assertion string and source file name for each assert that may fail. It is still possible to find the failed assert in the code by looking at the memory address where the assertion failed.
- Besides the `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL`, you can disable or silent the assertion for the HAL component separately by setting `CONFIG_HAL_DEFAULT_ASSERTION_LEVEL`. It is to notice that ESP-IDF lowers the HAL assertion level in bootloader to be silent even if `CONFIG_HAL_DEFAULT_ASSERTION_LEVEL` is set to full-assertion level. This is to reduce the bootloader size.
- Setting `CONFIG_COMPILER_OPTIMIZATION_CHECKS_SILENT` removes specific error messages for particular internal ESP-IDF error check macros. This may make it harder to debug some error conditions by reading the log output.
- Do not enable `CONFIG_COMPILER_CXX_EXCEPTIONS`, `CONFIG_COMPILER_CXX_RTTI`, or set the `CONFIG_COMPILER_STACK_CHECK_MODE` to Overall. All of these options are already disabled by default, but they have a large impact on binary size.
• Disabling `CONFIG_ESP_ERR_TO_NAME_LOOKUP` removes the lookup table to translate user-friendly names for error values (see Error Handling) in error logs, etc. This saves some binary size, but error values will be printed as integers only.

• Setting `CONFIG_ESP_SYSTEM_PANIC` to Silent reboot saves a small amount of binary size, however this is only recommended if no one will use UART output to debug the device.

• If the application binary uses only one of the security versions of the protocomm component, then the support for others can be disabled to save some code size. The support can be disabled through `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_0`, `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_1` or `CONFIG_ESP_PROTOCOMM_SUPPORT_SECURITY_VERSION_2` respectively.

**Note:** In addition to the many configuration items shown here, there are a number of configuration options where changing the option from the default increases binary size. These are not noted here. Where the increase is significant is usually noted in the configuration item help text.

**Targeted Optimizations** The following binary size optimizations apply to a particular component or a function:

**Wi-Fi**

• Disabling `CONFIG_ESP_WIFI_ENABLE_WPA3_SAE` will save some Wi-Fi binary size if WPA3 support is not needed. Note that WPA3 is mandatory for new Wi-Fi device certifications.

• Disabling `CONFIG_ESP_WIFI_SOFTAP_SUPPORT` will save some Wi-Fi binary size if soft-AP support is not needed.

• Disabling `CONFIG_ESP_WIFI_ENTERPRISE_SUPPORT` will save some Wi-Fi binary size if enterprise support is not needed.

**lwIP IPv6**

• Setting `CONFIG_LWIP_IPV6` to `false` will reduce the size of the lwIP TCP/IP stack, at the cost of only supporting IPv4.

**Note:** IPv6 is required by some components such as CoAP and ASIO Port. These components will not be available if IPv6 is disabled.

**lwIP IPv4**

• If IPv4 connectivity is not required, setting `CONFIG_LWIP_IPV4` to `false` will reduce the size of the lwIP, supporting IPv6-only TCP/IP stack.

**Note:** Before disabling IPv4 support, please note that IPv6 only network environments are not ubiquitous and must be supported in the local network, e.g., by your internet service provider or using constrained local network settings.

**Newlib Nano Formatting** By default, ESP-IDF uses Newlib “full” formatting for I/O functions (`printf()`, `scanf()`, etc.)

Enabling the config option `CONFIG_NEWLIB_NANO_FORMAT` will switch Newlib to the “Nano” formatting mode. This is smaller in code size, and a large part of the implementation is compiled into the ESP32-S2 ROM, so it does not need to be included in the binary at all.

The exact difference in binary size depends on which features the firmware uses, but 25 KB ~ 50 KB is typical.

Enabling “Nano” formatting reduces the stack usage of each function that calls `printf()` or another string formatting function, see Reducing Stack Sizes.
“Nano” formatting does not support 64-bit integers, or C99 formatting features. For a full list of restrictions, search for `--enable-newlib-nano-formatted-io` in the Newlib README file.

**MbedTLS Features** Under Component Config > mbedTLS, there are multiple mbedTLS features enabled by default, some of which can be disabled if not needed to save code size.

These include:

- `CONFIG_MBEDTLS_HAVE_TIME`
- `CONFIG_MBEDTLS_ECDSA_DETERMINISTIC`
- `CONFIG_MBEDTLS_SHA512_C`
- `CONFIG_MBEDTLS_CLIENT_SSL_SESSION_TICKETS`
- `CONFIG_MBEDTLS_SSL_CONTEXT.Serialization`
- `CONFIG_MBEDTLS_SSL_ALPN`
- `CONFIG_MBEDTLS_SSL_RENEGOTIATION`
- `CONFIG_MBEDTLS_CC M_C`
- `CONFIG_MBEDTLS_GCM_C`
- `CONFIG_MBEDTLS_ECP_C` (Alternatively: Leave this option enabled but disable some of the elliptic curves listed in the sub-menu.)
- `CONFIG_MBEDTLS_ECP_NIST_OPTIM`
- `CONFIG_MBEDTLS_ECP_FIXED_POINT_OPTIM`
- Change `CONFIG_MBEDTLS_TLS.MODE` if both server & client functionalities are not needed
- Consider disabling some cipher suites listed in the TLS Key Exchange Methods sub-menu (i.e., `CONFIG_MBEDTLS_KEY.EXCHANGE_RSA`)

The help text for each option has some more information for reference.

**Important:** It is strongly not recommended to disable all these mbedTLS options. Only disable options of which you understand the functionality and are certain that it is not needed in the application. In particular:

- Ensure that any TLS server(s) the device connects to can still be used. If the server is controlled by a third party or a cloud service, it is recommended to ensure that the firmware supports at least two of the supported cipher suites in case one is disabled in a future update.
- Ensure that any TLS client(s) that connect to the device can still connect with supported/recommended cipher suites. Note that future versions of client operating systems may remove support for some features, so it is recommended to enable multiple supported cipher suites, or algorithms for redundancy.

If depending on third party clients or servers, always pay attention to announcements about future changes to supported TLS features. If not, the ESP32-S2 device may become inaccessible if support changes.

**Note:** Not every combination of mbedTLS compile-time config is tested in ESP-IDF. If you find a combination that fails to compile or function as expected, please report the details on GitHub.

**VFS** Virtual Filesystem Component feature in ESP-IDF allows multiple filesystem drivers and file-like peripheral drivers to be accessed using standard I/O functions (`open, read, write, etc.`) and C library functions (`fopen, fread, fwrite, etc.`). When filesystem or file-like peripheral driver functionality is not used in the application, this feature can be fully or partially disabled. VFS component provides the following configuration options:

- `CONFIG_VFS_SUPPORT_TERMIOS` — can be disabled if the application does not use `termios` family of functions. Currently, these functions are implemented only for UART VFS driver. Most applications can disable this option. Disabling this option reduces the code size by about 1.8 KB.
- `CONFIG_VFS_SUPPORT_SELECT` — can be disabled if the application does not use the `select` function with file descriptors. Currently, only the UART and eventfd VFS drivers implement `select` support. Note that when this option is disabled, `select` can still be used for socket file descriptors. Disabling this option reduces the code size by about 2.7 KB.
• **CONFIG_VFS_SUPPORT_DIR** — can be disabled if the application does not use directory-related functions, such as `readdir` (see the description of this option for the complete list). Applications that only open, read and write specific files and do not need to enumerate or create directories can disable this option, reducing the code size by 0.5 KB or more, depending on the filesystem drivers in use.

• **CONFIG_VFS_SUPPORT_IO** — can be disabled if the application does not use filesystems or file-like peripheral drivers. This disables all VFS functionality, including the three options mentioned above. When this option is disabled, `Console` can not be used. Note that the application can still use standard I/O functions with socket file descriptors when this option is disabled. Compared to the default configuration, disabling this option reduces code size by about 9.4 KB.

**Bootloader Size**  
This document deals with the size of an ESP-IDF app binary only, and not the ESP-IDF Second Stage Bootloader.

For a discussion of ESP-IDF bootloader binary size, see Bootloader Size.

**IRAM Binary Size**  
If the IRAM section of a binary is too large, this issue can be resolved by reducing IRAM memory usage. See Optimizing IRAM Usage.

**Minimizing RAM Usage**

In some cases, a firmware application’s available RAM may run low or run out entirely. In these cases, it is necessary to tune the memory usage of the firmware application.

In general, firmware should aim to leave some headroom of free internal RAM to deal with extraordinary situations or changes in RAM usage in future updates.

**Background**  
Before optimizing ESP-IDF RAM usage, it is necessary to understand the basics of ESP32-S2 memory types, the difference between static and dynamic memory usage in C, and the way ESP-IDF uses stack and heap. This information can all be found in Heap Memory Allocation.

**Measuring Static Memory Usage**  
The `idf.py` tool can be used to generate reports about the static memory usage of an application, see Measuring Static Sizes.

**Measuring Dynamic Memory Usage**  
ESP-IDF contains a range of heap APIs for measuring free heap at runtime, see Heap Memory Debugging.

**Note:** In embedded systems, heap fragmentation can be a significant issue alongside total RAM usage. The heap measurement APIs provide ways to measure the largest free block. Monitoring this value along with the total number of free bytes can give a quick indication of whether heap fragmentation is becoming an issue.

**Reducing Static Memory Usage**

• Reducing the static memory usage of the application increases the amount of RAM available for heap at runtime, and vice versa.

• Generally speaking, minimizing static memory usage requires monitoring the `.data` and `.bss` sizes. For tools to do this, see Measuring Static Sizes.

• Internal ESP-IDF functions do not make heavy use of static RAM in C. In many instances (such as Wi-Fi library), static buffers are still allocated from the heap. However, the allocation is performed only once during feature initialization and will be freed if the feature is deinitialized. This approach is adopted to optimize the availability of free memory at various stages of the application’s life cycle.
To minimize static memory use:

- Constant data can be stored in flash memory instead of RAM, thus it is recommended to declare structures, buffers, or other variables as `const`. This approach may require modifying firmware functions to accept `const *` arguments instead of mutable pointer arguments. These changes can also help reduce the stack usage of certain functions.
- If using OpenThread, enabling the option `CONFIG_OPENTHREAD_PLATFORM_MSGPOOL_MANAGEMENT` will cause OpenThread to allocate message pool buffers from PSRAM, which will reduce static memory use.

### Reducing Stack Sizes

In FreeRTOS, task stacks are usually allocated from the heap. The stack size for each task is fixed and passed as an argument to `xTaskCreate()`. Each task can use up to its allocated stack size, but using more than this will cause an otherwise valid program to crash, with a stack overflow or heap corruption.

Therefore, determining the optimum sizes of each task stack, minimizing the required size of each task stack, and minimizing the number of task stacks as whole, can all substantially reduce RAM usage.

To determine the optimum size for a particular task stack, users can consider the following methods:

- At runtime, call the function `uxTaskGetStackHighWaterMark()` with the handle of any task where you think there is unused stack memory. This function returns the minimum lifetime free stack memory in bytes.  
  - The easiest time to call `uxTaskGetStackHighWaterMark()` is from the task itself: call `uxTaskGetStackHighWaterMark(NULL)` to get the current task’s high water mark after the time that the task has achieved its peak stack usage, i.e., if there is a main loop, execute the main loop a number of times with all possible states, and then call `uxTaskGetStackHighWaterMark()`.
  - Often, it is possible to subtract almost the entire value returned here from the total stack size of a task, but allow some safety margin to account for unexpected small increases in stack usage at runtime.
- Call `uxTaskGetSystemState()` at runtime to get a summary of all tasks in the system. This includes their individual stack high watermark values.
- When debugger watchpoints are not being used, users can set the `CONFIG_FREERTOS_WATCHPOINT_END_OF_STACK` option. This will cause one of the watchpoints to watch the last word of the task’s stack. If that word is overwritten (such as in a stack overflow), a panic is triggered immediately. This is slightly more reliable than the default `CONFIG_FREERTOS_CHECK_STACKOVERFLOW` option of Check using canary bytes, because the panic happens immediately, rather than on the next RTOS context switch. Neither option is perfect. In some cases, it is possible that the stack pointer skips the watchpoint or canary bytes and corrupts another region of RAM instead.

To reduce the required size of a particular task stack, users can consider the following methods:

- Avoid stack heavy functions. String formatting functions (like `printf()`) are particularly heavy users of the stack, so any task which does not ever call these can usually have its stack size reduced.
  - Enabling Newlib Nano Formatting reduces the stack usage of any task that calls `printf()` or other C string formatting functions.
- Avoid allocating large variables on the stack. In C, any large structures or arrays allocated as an automatic variable (i.e., default scope of a C declaration) uses space on the stack. To minimize the sizes of these, allocate them statically and/or see if you can save memory by dynamically allocating them from the heap only when they are needed.
- Avoid deep recursive function calls. Individual recursive function calls do not always add a lot of stack usage each time they are called, but if each function includes large stack-based variables then the overhead can get quite high.

To reduce the total number of tasks, users can consider the following method:

- Combine tasks. If a particular task is never created, the task’s stack is never allocated, thus reducing RAM usage significantly. Unnecessary tasks can typically be removed if those tasks can be combined with another task. In an application, tasks can typically be combined or removed if:
  - The work done by the tasks can be structured into multiple functions that are called sequentially.
  - The work done by the tasks can be structured into smaller jobs that are serialized (via a FreeRTOS queue or similar) for execution by a worker task.
Internal Task Stack Sizes

ESP-IDF allocates a number of internal tasks for housekeeping purposes or operating system functions. Some are created during the startup process, and some are created at runtime when particular features are initialized.

The default stack sizes for these tasks are usually set conservatively high to allow all common usage patterns. Many of the stack sizes are configurable, and it may be possible to reduce them to match the real runtime stack usage of the task.

**Important:** If internal task stack sizes are set too small, ESP-IDF will crash unpredictably. Even if the root cause is task stack overflow, this is not always clear when debugging. It is recommended that internal stack sizes are only reduced carefully (if at all), with close attention to high water mark free space under load. If reporting an issue that occurs when internal task stack sizes have been reduced, please always include the following information and the specific configuration that is being used.

- **Running the Main Task** has stack size `CONFIG_ESP_MAIN_TASK_STACK_SIZE`.
- **High Resolution Timer (ESP Timer)** system task which executes callbacks has stack size `CONFIG_ESP_TIMER_TASK_STACK_SIZE`.
- FreeRTOS Timer Task to handle FreeRTOS timer callbacks has stack size `CONFIG_FREERTOS_TIMER_TASK_STACK_SIZE`.
- **Event Loop Library** system task to execute callbacks for the default system event loop has stack size `CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE`.
- lwIP TCP/IP task has stack size `CONFIG_LWIP_TCPIP_TASK_STACK_SIZE`.
- The Ethernet driver creates a task for the MAC to receive Ethernet frames. If using the default config `ETH_MAC_DEFAULT_CONFIG` then the task stack size is 4 KB. This setting can be changed by passing a custom `eth_mac_config_t` struct when initializing the Ethernet MAC.
- FreeRTOS idle task stack size is configured by `CONFIG_FREERTOS_IDLE_TASK_STACKSIZE`.
- If using the **ESP-MQTT** component, it creates a task with stack size configured by `CONFIG_MQTT_TASK_STACK_SIZE`. MQTT stack size can also be configured using `task_stack` field of `esp_mqtt_client_config_t`.
- To see how to optimize RAM usage when using mDNS, please check Minimizing RAM Usage.

**Note:** Aside from built-in system features such as ESP-timer, if an ESP-IDF feature is not initialized by the firmware, then no associated task is created. In those cases, the stack usage is zero, and the stack-size configuration for the task is not relevant.

Reducing Heap Usage

For functions that assist in analyzing heap usage at runtime, see Heap Memory Debugging.

Normally, optimizing heap usage consists of analyzing the usage and removing calls to `malloc()` that are not being used, reducing the corresponding sizes, or freeing previously allocated buffers earlier.

There are some ESP-IDF configuration options that can reduce heap usage at runtime:

- lwIP documentation has a section to configure Minimum RAM Usage.
- Wi-Fi Buffer Usage describes options to either reduce the number of static buffers or reduce the maximum number of dynamic buffers in use, so as to minimize memory usage at a possible cost of performance. Note that static Wi-Fi buffers are still allocated from the heap when Wi-Fi is initialized, and will be freed if Wi-Fi is deinitialized.
- Several Mbed TLS configuration options can be used to reduce heap memory usage. See the Reducing Heap Usage docs for details.

**Note:** There are other configuration options that increases heap usage at runtime if changed from the defaults. These options are not listed above, but the help text for the configuration item will mention if there is some memory impact.
Optimizing IRAM Usage  The available DRAM at runtime for heap usage is also reduced by the static IRAM usage. Therefore, one way to increase available DRAM is to reduce IRAM usage.

If the app allocates more static IRAM than available, then the app will fail to build, and linker errors such as section '.iram0.text' will not fit in region 'iram0_0_seg', IRAM0 segment data does not fit, and region 'iram0_0_seg' overflowed by 84-bytes will be seen. If this happens, it is necessary to find ways to reduce static IRAM usage in order to link the application.

To analyze the IRAM usage in the firmware binary, use Measuring Static Sizes. If the firmware failed to link, steps to analyze are shown at Showing Size When Linker Fails.

The following options will reduce IRAM usage of some ESP-IDF features:

- Enable `CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH`. Provided these functions are not incorrectly used from ISRs, this option is safe to enable in all configurations.
- Enable `CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH`. Provided these functions are not incorrectly used from ISRs, this option is safe to enable in all configurations.
- Enable `CONFIG_RINGBUF_PLACE_ISR_FUNCTIONS_INTO_FLASH`. This option is not safe to use if the ISR ringbuf functions are used from an IRAM interrupt context, e.g., if `CONFIG_UART_ISR_IN_IRAM` is enabled. For the ESP-IDF drivers where this is the case, you can get an error at run-time when installing the driver in question.
- Disabling Wi-Fi options `CONFIG_ESP_WIFI_IRAM_OPT` and/or `CONFIG_ESP_WIFI_RX_IRAM_OPT` options frees available IRAM at the cost of Wi-Fi performance.
- Disabling `CONFIG_ESP_EVENT_POST_FROM_IRAM_ISR` prevents posting esp_event events from IRAM-Safe Interrupt Handlers but saves some IRAM.
- Disabling `CONFIG_SPI_MASTER_ISR_IN_IRAM` prevents spi_master interrupts from being serviced while writing to flash, and may otherwise reduce spi_master performance, but saves some IRAM.
- Disabling `CONFIG_SPI_SLAVE_ISR_IN_IRAM` prevents spi_slave interrupts from being serviced while writing to flash, which saves some IRAM.
- Setting `CONFIG_HAL_DEFAULT_ASSERTION_LEVEL` to disable assertion for HAL component saves some IRAM, especially for HAL code which calls `HAL_ASSERT` a lot and resides in IRAM.
- Refer to the sdkconfig menu Auto-detect Flash chips, and you can disable flash drivers which you do not need to save some IRAM.
- Enable `CONFIG_HEAP_PLACE_FUNCTION_INTO_FLASH`. Provided that `CONFIG_SPI_MASTER_ISR_IN_IRAM` is not enabled and the heap functions are not incorrectly used from ISRs, this option is safe to enable in all configurations.

**Note:** Moving frequently-called functions from IRAM to flash may increase their execution time.

**Note:** Other configuration options exist that will increase IRAM usage by moving some functionality into IRAM, usually for performance, but the default option is not to do this. These are not listed here. The IRAM size impact of enabling these options is usually noted in the configuration item help text.

### 4.24 Reproducible Builds

#### 4.24.1 Introduction

ESP-IDF build system has support for reproducible builds.

When reproducible builds are enabled, the application built with ESP-IDF does not depend on the build environment. Both the .elf file and .bin files of the application remains exactly the same, even if the following variables change:

- Directory where the project is located
- Directory where ESP-IDF is located (`IDF_PATH`)
- Build time
4.24.2 Reasons for Non-Reproducible Builds

There are several reasons why an application may depend on the build environment, even when the same source code and tools versions are used.

- In C code, __FILE__ preprocessor macro is expanded to the full path of the source file.
- __DATE__ and __TIME__ preprocessor macros are expanded to compilation date and time.
- When the compiler generates object files, it adds sections with debug information. These sections help debuggers, like GDB, to locate the source code which corresponds to a particular location in the machine code. These sections typically contain paths of relevant source files. These paths may be absolute, and will include the path to ESP-IDF or to the project.

There are also other possible reasons, such as unstable order of inputs and non-determinism in the build system.

4.24.3 Enabling Reproducible Builds in ESP-IDF

Reproducible builds can be enabled in ESP-IDF using CONFIG_APP_REPRODUCIBLE_BUILD option. This option is disabled by default. It can be enabled in menuconfig.

The option may also be added into sdkconfig.defaults. If adding the option into sdkconfig.defaults, delete the sdkconfig file and run the build again. See Custom Sdkconfig Defaults for more information.

4.24.4 How Reproducible Builds Are Achieved

ESP-IDF achieves reproducible builds using the following measures:

- In ESP-IDF source code, __DATE__ and __TIME__ macros are not used when reproducible builds are enabled. Note, if the application source code uses these macros, the build will not be reproducible.
- ESP-IDF build system passes a set of -fmacro-prefix-map and -fdebug-prefix-map flags to replace base paths with placeholders:
  - Path to ESP-IDF is replaced with /IDF
  - Path to the project is replaced with /IDF_PROJECT
  - Path to the build directory is replaced with /IDF_BUILD
  - Paths to components are replaced with /COMPONENT_NAME_DIR (where NAME is the name of the component)
- Build date and time are not included into the application metadata structure and bootloader metadata structure if CONFIG_APP_REPRODUCIBLE_BUILD is enabled.
- ESP-IDF build system ensures that source file lists, component lists and other sequences are sorted before passing them to CMake. Various other parts of the build system, such as the linker script generator also perform sorting to ensure that same output is produced regardless of the environment.

4.24.5 Reproducible Builds and Debugging

When reproducible builds are enabled, file names included in debug information sections are altered as shown in the previous section. Due to this fact, the debugger (GDB) is not able to locate the source files for the given code location.

This issue can be solved using GDB set substitute-path command. For example, by adding the following command to GDB init script, the altered paths can be reverted to the original ones:

```
set substitute-path /COMPONENT_FREERTOS_DIR /home/user/esp/esp-idf/components/freertos
```

ESP-IDF build system generates a file with the list of such set substitute-path commands automatically during the build process. The file is called prefix_map_gdbinit and is located in the project build directory.

When idf.py gdb is used to start debugging, this additional gdbinit file is automatically passed to GDB. When launching GDB manually or from and IDE, please pass this additional gdbinit script to GDB using -x build/prefix_map_gdbinit argument.
4.24.6 Factors Which Still Affect Reproducible Builds

Note that the built application still depends on:

- ESP-IDF version
- Versions of the build tools (CMake, Ninja) and the cross-compiler

*IDF Docker Image* can be used to ensure that these factors do not affect the build.

4.25 RF Calibration

ESP32-S2 supports three RF calibration methods during RF initialization:

1. Partial calibration
2. Full calibration
3. No calibration

4.25.1 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_ESP_PHY_CALIBRATION_AND_DATA_STORAGE`.

4.25.2 Full Calibration

Full calibration is triggered in the following conditions:

1. NVS does not exist.
2. The NVS partition to store calibration data has been erased.
3. Hardware MAC address has changed.
4. PHY library version has changed.
5. The RF calibration data loaded from the NVS partition is broken.

Full calibration takes 100 ms longer than the partial calibration method. If boot duration is not of critical importance to the application, the full calibration method is recommended. To switch to the full calibration method, go to *menuconfig* and disable `CONFIG_ESP_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 do not 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 Wi-Fi and Bluetooth®/Bluetooth Low Energy based on some conditions (e.g., an option provided in some diagnostic mode). In this case, only the PHY namespace of the NVS partition is erased.

4.25.3 No Calibration

The no calibration method is only used when the device wakes up from Deep-sleep mode.

4.25.4 PHY Initialization Data

The PHY initialization data is used for RF calibration. There are two ways to get the PHY initialization data.

One is to use the default initialization data which is located in the header file `components/esp_phy/esp32s2/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_ESP_PHY_INIT_DATA_IN_PARTITION.

An alternative is to store the initialization data in a PHY data partition. A PHY data partition is included in the default partition table. However, when using a custom partition table, please ensure that a PHY data partition (type: data, subtype: phy) is included in the custom partition table. Whether you are using a custom partition table or the default partition table, if initialization data is stored in a partition, it has to be flashed there, otherwise a runtime error occurs. If you want to use initialization data stored in a partition, go to menuconfig and enable the option CONFIG_ESP_PHY_INIT_DATA_IN_PARTITION.

4.25.5 API Reference

Header File

• components/esp_phy/include/esp_phy_init.h
  • This header file can be included with:
    ```
    #include "esp_phy_init.h"
    ```
  • This header file is a part of the API provided by the esp_phy component. To declare that your component depends on esp_phy, add the following to your CMakeLists.txt:
    ```
    REQUIRES esp_phy
    ```
    or
    ```
    PRIV_REQUIRES esp_phy
    ```

Functions

```c
const esp_phy_init_data_t *esp_phy_get_init_data(void)
```

Get PHY init data.

If “Use a partition to store PHY init data” option is set in menuconfig, this function will load PHY init data from a partition. Otherwise, PHY init data will be compiled into the application itself, and this function will return a pointer to PHY init data located in read-only memory (DROM).

If “Use a partition to store PHY init data” option is enabled, this function may return NULL if the data loaded from flash is not valid.

**Note:** Call esp_phy_release_init_data to release the pointer obtained using this function after the call to esp_wifi_init.

```c
void esp_phy_release_init_data(const esp_phy_init_data_t *data)
```

Release PHY init data.

```c
esp_err_t esp_phy_load_cal_data_from_nvs(esp_phy_calibration_data_t *out_cal_data)
```

Function called by esp_phy_load_cal_and_init to load PHY calibration data.

This is a convenience function which can be used to load PHY calibration data from NVS. Data can be stored to NVS using esp_phy_store_cal_data_to_nvs function.

If calibration data is not present in the NVS, or data is not valid (was obtained for a chip with a different MAC address, or obtained for a different version of software), this function will return an error.
Chapter 4. API Guides

Parameters **out_cal_data** – pointer to calibration data structure to be filled with loaded data.

Returns ESP_OK on success

```c
esp_err_t esp_phy_store_cal_data_to_nvs (const esp_phy_calibration_data_t *cal_data)
```

Function called by esp_phy_load_cal_and_init to store PHY calibration data.

This is a convenience function which can be used to store PHY calibration data to the NVS. Calibration data is returned by esp_phy_load_cal_and_init function. Data saved using this function to the NVS can later be loaded using esp_phy_store_cal_data_to_nvs function.

Parameters **cal_data** – pointer to calibration data which has to be saved.

Returns ESP_OK on success

```c
esp_err_t esp_phy_erase_cal_data_in_nvs (void)
```

Erase PHY calibration data which is stored in the NVS.

This is a function which can be used to trigger full calibration as a last-resort remedy if partial calibration is used. It can be called in the application based on some conditions (e.g. an option provided in some diagnostic mode).

Returns ESP_OK on success

Returns others on fail. Please refer to NVS API return value error number.

```c
void esp_phy_enable (esp_phy_modem_t modem)
```

Enable PHY and RF module.

PHY and RF module should be enabled in order to use WiFi or BT. Now PHY and RF enabling job is done automatically when start WiFi or BT. Users should not call this API in their application.

Parameters **modem** – the modem to call the phy enable.

```c
void esp_phy_disable (esp_phy_modem_t modem)
```

Disable PHY and RF module.

PHY module should be disabled in order to shutdown WiFi or BT. Now PHY and RF disabling job is done automatically when stop WiFi or BT. Users should not call this API in their application.

Parameters **modem** – the modem to call the phy disable.

```c
void esp_btbb_enable (void)
```

Enable BTBB module.

BTBB module should be enabled in order to use IEEE802154 or BT. Now BTBB enabling job is done automatically when start IEEE802154 or BT. Users should not call this API in their application.

```c
void esp_btbb_disable (void)
```

Disable BTBB module, used by IEEE802154 or Bluetooth. Users should not call this API in their application.

```c
void esp_phy_load_cal_and_init (void)
```

Load calibration data from NVS and initialize PHY and RF module.

```c
void esp_phy_modem_init (void)
```

Initialize backup memory for Phy power up/down.

```c
void esp_phy_modem_deinit (void)
```

Deinitialize backup memory for Phy power up/down Set phy_init_flag if all modems deinit on ESP32C3.

```c
void esp_phy_common_clock_enable (void)
```

Enable WiFi/BT common clock.

```c
void esp_phy_common_clock_disable (void)
```

Disable WiFi/BT common clock.
int64_t esp_phy_rf_get_on_ts (void)
   Get the time stamp when PHY/RF was switched on.

   Returns return 0 if PHY/RF is never switched on. Otherwise return time in microsecond since
   boot when phy/rf was last switched on

esp_err_t esp_phy_update_country_info (const char *country)
   Update the corresponding PHY init type according to the country code of Wi-Fi.

   Parameters country -country code
   Returns ESP_OK on success.
   Returns esp_err_t code describing the error on fail

char *get_phy_version_str (void)
   Get PHY lib version.

void phy_init_param_set (uint8_t param)
   Set PHY init parameters.

   Parameters param --is 1 means combo module

void phy_wifi_enable_set (uint8_t enable)
   Wi-Fi RX enable.

   Parameters enable --True for enable wifi receiving mode as default, false for closing wifi re-
   ceiving mode as default.

**Structures**

struct esp_phy_init_data_t
   Structure holding PHY init parameters.

   **Public Members**

   uint8_t params[128]
      opaque PHY initialization parameters

struct esp_phy_calibration_data_t
   Opaque PHY calibration data.

   **Public Members**

   uint8_t version[4]
      PHY version

   uint8_t mac[6]
      The MAC address of the station

   uint8_t opaque[1894]
      calibration data
Enumerations

define esp_phy_modem_t
    PHY enable or disable modem.
    \n    Values:
    \n    enumerator PHY_MODEM_WIFI
        PHY modem WiFi
    enumerator PHY_MODEM_BT
        PHY modem BT
    enumerator PHY_MODEM_IEEE802154
        PHY modem IEEE802154

define esp_phy_calibration_mode_t
    PHY calibration mode.
    \n    Values:
    \n    enumerator PHY_RF_CAL_PARTIAL
        Do part of RF calibration. This should be used after power-on reset.
    enumerator PHY_RF_CAL_NONE
        Don’t do any RF calibration. This mode is only suggested to be used after deep sleep reset.
    enumerator PHY_RF_CAL_FULL
        Do full RF calibration. Produces best results, but also consumes a lot of time and current. Suggested to be used once.

Header File

- components/esp_phy/include/esp_phy_cert_test.h
- This header file can be included with:

```c
#include "esp_phy_cert_test.h"
```

- This header file is a part of the API provided by the esp_phy component. To declare that your component depends on esp_phy, add the following to your CMakeLists.txt:

```cmake
REQUIRES esp_phy
```
or

```cmake
PRIV_REQUIRES esp_phy
```

Functions

void esp_wifi_power_domain_on (void)
    Wifi power domain power on.

void esp_wifi_power_domain_off (void)
    Wifi power domain power off.
void esp_phy_rf_test_config (uint8_t conf)
    Environment variable configuration.
    **Parameters**
    - **conf** – Set to 1 to enter RF test mode.

void esp_phy_rf_test_init (void)
    RF initialization configuration.

void esp_phy_tx_contin_en (bool contin_en)
    TX Continuous mode.
    **Parameters**
    - **contin_en** – Set to true for continuous packet sending, which can be used for certification testing; Set to false to cancel continuous mode, which is the default mode and can be used for WLAN tester.

void esp_phy_cbw_40m_en (bool en)
    HT40/HT20 mode selection.
    **Parameters**
    - **en** – Set to false to enter 11n HT20 mode; Set to true to enter 11n HT40 mode;

void esp_phy_wifi_tx (uint32_t chan, esp_phy_wifi_rate_t rate, int8_t backoff, uint32_t length_byte, uint32_t packet_delay, uint32_t packet_num)
    Wi-Fi TX command.
    **Parameters**
    - **chan** – channel setting, 1~14;
    - **rate** – rate setting;
    - **backoff** – Transmit power attenuation, unit is 0.25dB. For example, 4 means that the power is attenuated by 1dB;
    - **length_byte** – TX packet length configuration, indicating PSDU Length, unit is byte;
    - **packet_delay** – TX packet interval configuration, unit is us;
    - **packet_num** – The number of packets sent, 0 means sending packets continuously, other values represent the number of packets to send.

void esp_phy_test_start_stop (uint8_t value)
    Test start/stop command, used to stop transmitting or receiving state.
    **Parameters**
    - **value** – Value should be set to 3 before TX/RX. Set value to 0 to end TX/RX state.

void esp_phy_wifi_rx (uint32_t chan, esp_phy_wifi_rate_t rate)
    Wi-Fi RX command.
    **Parameters**
    - **chan** – channel setting, 1~14;
    - **rate** – rate setting;

void esp_phy_wifi_tx_tone (uint32_t start, uint32_t chan, uint32_t backoff)
    Wi-Fi Carrier Wave(CW) TX command.
    **Parameters**
    - **start** – enable CW, 1 means transmit, 0 means stop transmitting;
    - **chan** – CW channel setting, 1~14;
    - **backoff** – CW power attenuation parameter, unit is 0.25dB. 4 indicates the power is attenuated by 1dB.

void esp_phy_ble_tx (uint32_t txpwr, uint32_t chan, uint32_t len, esp_phy_ble_type_t data_type, uint32_t syncw, esp_phy_ble_rate_t rate, uint32_t tx_num_in)
    BLE TX command.
    **Parameters**
    - **txpwr** – Transmit power level. Tx power is about (level-8)*3 dBm, step is 3dB. Level 8 is around 0 dBm;
    - **chan** – channel setting, range is 0~39, corresponding frequency = 2402+chan*2;
    - **len** – Payload length setting, range is 0-255, unit is byte, 37 bytes is employed generally;
• data_type – Data type setting;
• syncw – Packet identification (need to be provided by the packet generator or instrument manufacturer), 0x71764129 is employed generally;
• rate – rate setting;
• tx_num_in – The number of packets sent, 0 means sending packets continuously, other values represent the number of packets to send.

void esp_phy_ble_rx (uint32_t chan, uint32_t syncw, esp_phy_ble_rate_t rate)
BLE RX command.

Parameters
• chan – channel selection, range is 0-39; Channels 0, 1, 2~10 correspond to 2404MHz, 2406MHz, 2408MHz~2424MHz respectively; Channels 11, 12, 13~36 correspond to 2428MHz, 2430MHz, 2432MHz~2478MHz respectively; Channel 37: 2402MHz, Channel 38: 2426MHz, Channel 39: 2480MHz;
• syncw – Packet identification (need to be provided by the packet generator or instrument manufacturer), 0x71764129 is employed generally;
• rate – rate setting;

void esp_phy_bt_tx_tone (uint32_t start, uint32_t chan, uint32_t power)
BLE Carrier Wave (CW) TX command.

Parameters
• start – enable CW, 1 means transmit, 0 means stop transmitting;
• chan – Single carrier transmission channel selection, range is 0~39, corresponding frequency freq = 2402+chan*2;
• power – CW power attenuation parameter, unit is 0.25dB. 4 indicates the power is attenuated by 1dB.

void esp_phy_get_rx_result (esp_phy_rx_result_t *rx_result)
Get some RX information.

Parameters rx_result – This struct for storing RX information;

Structures

struct esp_phy_rx_result_t
Structure holding PHY RX result.

Public Members

uint32_t phy_rx_correct_count
The number of desired packets received

int phy_rx_rssi
Average RSSI of desired packets

uint32_t phy_rx_total_count
The number of total packets received

uint32_t phy_rx_result_flag
0 means no RX info; 1 means the lastest Wi-Fi RX info; 2 means the lastest BLE RX info.
Enumerations

```c
enum esp_phy_wifi_rate_t
{
  PHY_RATE_1M,
  PHY_RATE_2M,
  PHY_RATE_5M5,
  PHY_RATE_11M,
  PHY_RATE_6M,
  PHY_RATE_9M,
  PHY_RATE_12M,
  PHY_RATE_18M,
  PHY_RATE_24M,
  PHY_RATE_36M,
  PHY_RATE_48M,
  PHY_RATE_54M,
  PHY_RATE_MCS0,
  PHY_RATE_MCS1,
  PHY_RATE_MCS2,
  PHY_RATE_MCS3,
  PHY_RATE_MCS4,
  PHY_RATE_MCS5,
  PHY_RATE_MCS6,
  PHY_RATE_MCS7,
  PHY_WIFI_RATE_MAX
};
```
enum esp_phy_ble_rate_t

Values:

enumerator PHY_BLE_RATE_1M
enumerator PHY_BLE_RATE_2M
enumerator PHY_BLE_RATE_125K
enumerator PHY_BLE_RATE_500k
enumerator PHY_BLE_RATE_MAX

enum esp_phy_ble_type_t

Values:

enumerator PHY_BLE_TYPE_1010
enumerator PHY_BLE_TYPE_00001111
enumerator PHY_BLE_TYPE_prbs9
enumerator PHY_BLE_TYPE_00111100
enumerator PHY_BLE_TYPE_MAX

4.26 Thread Local Storage

4.26.1 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 APIs: ESP-IDF FreeRTOS native APIs.
• Pthread APIs: ESP-IDF pthread APIs.
• C11 Standard: C11 standard introduces special keywords to declare variables as thread local.

4.26.2 FreeRTOS Native APIs

The ESP-IDF FreeRTOS provides the following APIs to manage thread local variables:

• vTaskSetThreadLocalStoragePointer()
• pvTaskGetThreadLocalStoragePointer()
• vTaskSetThreadLocalStoragePointerAndDelCallback()
In this case, the maximum number of variables that can be allocated is limited by `CONFIG_FREERTOS_THREAD_LOCAL_STORAGE_POINTERS`. 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 the APIs above, you 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 you need to allocate/deallocate memory for it. Variable’s deallocation is initiated by FreeRTOS when task is deleted, but user must provide callback function to do proper cleanup.

### 4.26.3 Pthread APIs

The ESP-IDF provides the following POSIX Threads Support to manage thread local variables:

- `pthread_key_create()`
- `pthread_key_delete()`
- `pthread_getspecific()`
- `pthread_setspecific()`

These APIs have all benefits of the ones above, but eliminates some their 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.

### 4.26.4 C11 Standard

The ESP-IDF FreeRTOS supports thread local variables according to C11 standard, ones specified with `__thread` keyword. For details on this feature, please refer to the GCC documentation.

Storage for that kind of variables is allocated on the task stack. Note that area for all such variables in the program is 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 (e.g., `ipc`, `timer` tasks etc.) will also have that extra stack space allocated. Thus feature should be used with care.

Using C11 thread local variables comes at a trade-off. On one hand, they are quite handy to use in programming and can be accessed using minimal CPU instructions. However, this benefit comes at 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.

### 4.27 Tools

#### 4.27.1 IDF Frontend - `idf.py`

The `idf.py` command-line tool provides a front-end for easily managing your project builds, deployment and debugging, and more. It configures several tools, for example:

- `CMake`, which configures the project to be built.
- `Ninja`, which builds the project.
- `esptool.py`, which flashes the target.

The Step 5. First Steps on ESP-IDF contains a brief introduction on how to set up `idf.py` to configure, build, and flash projects.

**Important:** `idf.py` should be run in an ESP-IDF project directory, i.e., a directory containing a `CMakeLists.txt` file. Older style projects that contain a `Makefile` will not work with `idf.py`. 
### Commands

**Start a New Project: create-project**

```bash
idf.py create-project <project name>
```

This command creates a new ESP-IDF project. Additionally, the folder where the project will be created in can be specified by the `--path` option.

**Create a New Component: create-component**

```bash
idf.py create-component <component name>
```

This command creates a new component, which will have a minimum set of files necessary for building. The `-C` option can be used to specify the directory the component will be created in. For more information about components see the Component CMakeLists Files.

**Select the Target Chip: set-target**

ESP-IDF supports multiple targets (chips). A full list of supported targets in your version of ESP-IDF can be seen by running `idf.py --list-targets`.

```bash
idf.py set-target <target>
```

This command sets the current project target.

**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 the `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=esp32s2`) or as a CMake variable (e.g., `-DIDF_TARGET=esp32s2` 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, please add the `CONFIG_IDF_TARGET` option to the project’s `sdkconfig.defaults` file, e.g., `CONFIG_IDF_TARGET="esp32s2"`. This value of the option will be used if `IDF_TARGET` is not specified by other methods, such as using an environment variable, a CMake variable, or the `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.

**Start the Graphical Configuration Tool: menuconfig**

```bash
idf.py menuconfig
```

**Build the Project: build**

```bash
idf.py build
```

This command builds 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. Additionally, the command can be run with app, bootloader and partition-table arguments to build only the app, bootloader or partition table as applicable.

**Remove the Build Output: clean**

```
idf.py clean
```

This command removes the project build output files from the build directory, and the project will be fully rebuilt on next build. Using this command does not remove the CMake configuration output inside the build folder.

**Delete the Entire Build Contents: fullclean**

```
idf.py fullclean
```

This command deletes the entire “build” directory contents, which 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.

**Flash the Project: flash**

```
idf.py flash
```

This command automatically builds the project if necessary, and then flash it to the target. You can use -p and -b options to set serial port name and flasher baud rate, respectively.

**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.

Similarly to the build command, the command can be run with app, bootloader and partition-table arguments to flash only the app, bootloader or partition table as applicable.

**Hints on How to Resolve Errors**

idf.py will try to suggest hints on how to resolve errors. It works with a database of hints stored in tools/idf_py_actions/hints.yml and the hints will be printed if a match is found for the given error. The menuconfig target is not supported at the moment by automatic hints on resolving errors.

The --no-hints argument of idf.py can be used to turn the hints off in case they are not desired.

**Important Notes**

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 target before running the serial monitor.

The order of multiple idf.py commands on the same invocation is not important, as they will automatically be executed in the correct order for everything to take effect (e.g., building before flashing, erasing before flashing).

For commands that are not known to idf.py, an attempt to execute them as a build system target will be made.

The command idf.py supports shell autocompletion for bash, zsh and fish shells.

To enable autocompletion for idf.py, use the export command (Step 4. Set up the environment variables). Autocompletion is initiated by pressing the TAB key. Type idf.py - and press the TAB key to autocomplete options.
Chapter 4  API Guides

The autocomplete support for PowerShell is planned in the future.

**Advanced Commands**

**Open the Documentation: docs**

```
idf.py docs
```

This command opens the documentation for the projects target and ESP-IDF version in the browser.

**Show Size: size**

```
idf.py size
```

This command prints app size information including the occupied RAM and flash and section (i.e., .bss) sizes.

```
idf.py size-components
```

Similarly, this command prints the same information for each component used in the project.

```
idf.py size-files
```

This command prints size information per source file in the project.

**Options**

- **--format** specifies the output format with available options: text, csv, json, default being text.
- **--output-file** optionally specifies the name of the file to print the command output to instead of the standard output.

**Reconfigure the Project: reconfigure**

```
idf.py reconfigure
```

This command forces CMake to be rerun regardless of whether it is necessary. It is unnecessary 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.

**Clean the Python Byte Code: python-clean**

```
idf.py python-clean
```

This command deletes generated python byte code from the ESP-IDF directory. The byte code may cause issues when switching between ESP-IDF and Python versions. It is advised to run this target after switching versions of Python.

**Generate a UF2 Binary: uf2**

```
idf.py uf2
```

This command generates a UF2 (USB Flashing Format) binary `uf2.bin` in the build directory. This file includes all the necessary binaries (bootloader, app, and partition table) for flashing the target.

This UF2 file can be copied to a USB mass storage device exposed by another ESP running the ESP USB Bridge project. The bridge MCU will use it to flash the target MCU. This is as simple as copying (or “drag-and-dropping”) the file to the exposed disk accessed by a file explorer in your machine.

To generate a UF2 binary for the application only (not including the bootloader and partition table), use the `uf2-app` command.
Global 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`, e.g., `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` enables CCache when compiling source files if the CCache tool is installed. This can dramatically reduce the build time.

**Important:** Note that some older versions of CCache may exhibit bugs on some platforms, so if files are not rebuilt as expected, try disabling CCache and rebuilding the project. To enable CCache by default, set the `IDF_CCACHE_ENABLE` environment variable to a non-zero value.

- `--cmake-warn-uninitialized` (or `-w`) causes CMake to print uninitialized variable warnings found in the project directory only. 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.
- `--no-hints` flag disables hints on resolving errors and disable capturing output.

### 4.27.2 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`, and `zip`.
- Python 3.8 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 the ESP-IDF location in the container.
- All the build tools required for the specific version of ESP-IDF: CMake, Ninja, cross-compiler toolchains, etc.
- All Python packages required by ESP-IDF are installed in a virtual environment.

The image `ENTRYPOINT` sets up the `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](https://hub.docker.com/r/espressif/idf/tags).
Chapter 4. API Guides

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 a 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 -u $UID -e HOME=/tmp 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.
- `-w /project`: makes `/project` the working directory for the command.
- `-u $UID`: makes the command run with your user ID so that files are created as you (instead of root).
- `-e HOME=/tmp`: gives the user a home directory for storing temporary files created by `idf.py` in `~/.cache`.
- `espressif/idf`: uses Docker image `espressif/idf` with tag `latest`. The `latest` tag is 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`, for example:

```
docker run --rm -v $PWD:/project -w /project -u $UID -e HOME=/tmp espressif/idf
     --idf:release-v4.4 idf.py build
```

You can check the up-to-date list of available tags at https://hub.docker.com/r/espressif/idf/tags.

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 -u $UID -e HOME=/tmp -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` does not work in the container, unless the serial port is passed through into the container. This can be done with Docker for Linux with the `device` option. 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). This limitation may be overcome by using remote serial ports. An example of how to do this can be found in the following `using remote serial port` section.

Using Remote Serial Port  The RFC2217 (Telnet) protocol can be used to remotely connect to a serial port. For more information please see the remote serial ports documentation in the ESP tool project. This method can also be used to access the serial port inside a Docker container if it cannot be accessed directly. Following is an example of how to use the Flash command from within a Docker container.

On host install and start `esp_rfc2217_server`:
• On Windows, the package is available as a one-file bundled executable created by **pyinstaller** and it can be downloaded from the **esptool releases** page in a ZIP archive along with other ESP tool utilities:

```
esp_rfc2217_server -v -p 4000 COM3
```

• On Linux or macOS, the package is available as part of **esptool**, which can be found in the ESP-IDF environment or by installing using **pip**:

```
pip install esptool
```

And then starting the server by executing

```
esp_rfc2217_server.py -v -p 4000 /dev/ttyUSB0
```

Now the device attached to the host can be flashed from inside a Docker container by using:

```
docker run --rm -v <host_path>:<container_path> -w /<container_path> espressif/\n-idf idf.py --port 'rfc2217://host.docker.internal:4000?ign_set_control' flash
```

Please make sure that `<host_path>` is properly set to your project path on the host, and `<container_path>` is set as a working directory inside the container with the `-w` option. The `host.docker.internal` is a special Docker DNS name to access the host. This can be replaced with a host IP if necessary.

### Building Custom Images

The Dockerfile in ESP-IDF repository provides several build arguments which can be used to customize the Docker image:

- **IDF_CLONE_URL**: URL of the repository to clone ESP-IDF from. Can be set to a custom URL when working with a fork of ESP-IDF. The default is `https://github.com/espressif/esp-idf.git`.
- **IDF_CLONE_BRANCH_OR_TAG**: Name of a git branch or tag used when cloning ESP-IDF. This value is passed to the `git clone` command using the `--branch` argument. The default is `master`.
- **IDF_CHECKOUT_REF**: If this argument is set to a non-empty value, `git checkout $IDF_CHECKOUT_REF` command performs after cloning. This argument can be set to the SHA of the specific commit to check out, for example, if some specific commit on a release branch is desired.
- **IDF_CLONE_SHALLOW**: If this argument is set to a non-empty value, `--depth=1 --shallow-submodules` arguments are be used when performing `git clone`. This significantly reduces the amount of data downloaded and the size of the resulting Docker image. However, if switching to a different branch in such a “shallow” repository is necessary, an additional `git fetch origin <branch>` command must be executed first.
- **IDF_INSTALL_TARGETS**: Comma-separated list of ESP-IDF targets to install toolchains for, or all to install toolchains for all targets. Selecting specific targets reduces the amount of data downloaded and the size of the resulting Docker image. The default is `all`.

To use these arguments, pass them via the `--build-arg` command line option. For example, the following command builds a Docker image with a shallow clone of ESP-IDF v4.4.1 and tools for ESP32-C3 only:

```
docker build -t idf-custom:v4.4.1-esp32c3 \  
--build-arg IDF_CLONE_BRANCH_OR_TAG=v4.4.1 \  
--build-arg IDF_CLONE_SHALLOW=1 \  
--build-arg IDF_INSTALL_TARGETS=esp32c3 \  
tools/docker
```

#### 4.27.3 IDF Windows Installer
**Command-Line Parameters**

Windows Installer `esp-idf-tools-setup` provides the following command-line parameters:

- /CONFIG=[PATH] - Path to ini configuration file to override default configuration of the installer. Default: config.ini.
- /GITRECURSIVE=[yes|no] - Clone recursively all Git repository submodules. Default: yes.
- /IDFDIR=[PATH] - Path to directory where it is installed. Default: `{userdesktop}\esp-idf`.
- /OFFLINE=[yes|no] - Execute installation of Python packages by pip in offline mode. The same result can be achieved by setting the environment variable PIP_NO_INDEX. Default: no.
- /PYTHONNOUSERSITE=[yes|no] - Set PYTHONNOUSERSITE variable before launching any Python command to avoid loading Python packages from AppDataRoaming. Default: yes.
- /PYTHONWHEELSURL=[URL] - Specify URLs to PyPi repositories for resolving binary Python Wheel dependencies. The same result can be achieved by setting the environment variable PIP_EXTRA_INDEX_URL. Default: https://dl.espressif.com/pypi.
- /VERYSILENT /SUPPRESSMSGBOXES /SP- /NOCANCEL - Perform silent installation.

**Unattended Installation**

The unattended installation of ESP-IDF can be achieved by following command-line parameters:

```
esp-idf-tools-setup-x.x.exe /VERYSILENT /SUPPRESSMSGBOXES /SP- /NOCANCEL
```

When running the installer from the command line, it detaches its process from the command line and starts a separate process in the background to perform the installation without blocking the use of the command line. The following PowerShell script allows you to wait for the installer to complete:

```
$InstallerProcess = Get-Process esp-idf-tools-setup
Wait-Process -Id $InstallerProcess.id
```

**Custom Python and Custom Location of Python Wheels**

The IDF installer is using by default embedded Python with reference to the Python Wheel mirror. The following parameters allow to select custom Python and custom location of Python wheels:

```
esp-idf-tools-setup-x.x.exe /USEEMBEDDEDPYTHON=no /PYTHONWHEELSURL=https://pypi.org/simple/
```

**4.27.4 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/.

For detailed information about the IDF Component Manager, see the IDF Component Manager and ESP Component Registry Documentation.

**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 by running `idf.py create-manifest`. By default, a manifest file is created for the main component. You can explicitly either specify the directory where the manifest should be created using the `--path` option or specify the component in your components folder using `--component=my_component`. The `create-manifest` command can be run in the following ways:

- `idf.py create-manifest` creates a manifest file for the main component
- `idf.py create-manifest --component=my_component` creates a manifest file for the component `my_component` in the components directory
- `idf.py create-manifest --path=".../my_component."` creates a manifest file for the component `my_component` in the `my_component` directory

When a new manifest is added to one of the components in the project, it is necessary to reconfigure the project manually by running `idf.py reconfigure`. The build will then track changes in `idf_component.yml` manifests and automatically trigger CMake when necessary.

To add a dependency to a component (e.g., `my_component`) in your ESP-IDF project, you can run the command `idf.py add-dependency DEPENDENCY`. The `DEPENDENCY` argument represents an additional component managed by the IDF Component Manager that `my_component` depends on. It is defined in the format `namespace/name=1.0.0`, where `namespace/name` is the name of the component and `=1.0.0` is a version range of the component, see the Versioning Documentation. By default, dependencies are added to the main component. You can either explicitly specify a directory where the manifest is located using the `--path` option, or specify the component in your `components` folder using `--component=my_component`. The `add-dependency` command can be run in the following ways:

- `idf.py add-dependency example/cmp` adds a dependency on the most recent version of `example/cmp` to the main component
- `idf.py add-dependency --component=my_component example/cmp<=3.3.3` adds a dependency on the version `<=3.3.3` of `example/cmp` to the component `my_component` in the components directory
- `idf.py add-dependency --path=".../my_component." example/cmp^3.3.3` adds a dependency on the version `^3.3.3` of `example/cmp` to the component `my_component` in the `my_component` directory

**Note:** The command `add-dependency` adds dependencies to your project explicitly from the Espressif Component Registry.

To update dependencies of the ESP-IDF project, you can run the command `idf.py update-dependencies`. You can also specify the path to the project directory using `--project-dir PATH`.

There is an example application `build_system/cmake/component_manager` that uses components installed by the component manager.

It is not necessary to have a manifest for components that do not 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.
Chapter 4. API Guides

- 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 the content of the managed_components directory are not supposed to be modified by a user. When the component manager runs, it always makes sure they are up to date. If these files were accidentally modified, it is possible to re-run the component manager by triggering CMake with idf.py reconfigure.

You may set the build property DEPENDENCIES_LOCK to specify the lock-file path in the top-level CMakeLists.txt. For example, adding `idf_build_set_property(DEPENDENCIES_LOCK dependencies.lock.$(IDF_TARGET))` before `project(PROJECT_NAME)` could help generate different lock files for different targets.

Creating a Project From an Example

Some components on the registry contain example projects. To create a new project from an example you can run the command `idf.py create-project-from-example EXAMPLE`. The EXAMPLE argument should be in the format `namespace/name=1.0.0:example` where `namespace/name` is the name of the component, `=1.0.0` is a version range of the component (see the Versioning Documentation) and `example` is the example’s name. You can find the list of examples for every component and the command to start a project for it in the Espressif Component Registry.

Defining Dependencies in the Manifest

You can easily define dependencies in the manifest file `idf_component.yml` by editing it directly in the text editor. Below are some basic examples that demonstrate how to define dependencies.

You can define a dependency from the registry by specifying the component name and the version range:

```yaml
dependencies:
  # Define a dependency from the registry (https://components.espressif.com/
  # component/example/cmp)
  example/cmp: ">=1.0.0"
```

To define a dependency from a Git repository, provide the path to the component within the repository and the repository’s URL:

```yaml
dependencies:
  # Define a dependency from a Git repository
test_component:
    path: test_component
    git: ssh://git@gitlab.com/user/components.git
```

During the development of components, you can use components from a local directory by specifying either a relative or an absolute path:

```yaml
dependencies:
  # Define local dependency with relative path
some_local_component:
  path: ../../projects/component
```

For detailed information about the manifest file format, see Manifest File Format Documentation.

Disabling the Component Manager

The component manager can be explicitly disabled by setting the IDF_COMPONENT_MANAGER environment variable to 0.
4.27.5 IDF Clang-Tidy

The IDF Clang Tidy is a tool that uses clang-tidy to run static analysis on your current app.

**Warning:** This functionality and the toolchain it relies on are still under development. There may be breaking changes before a final release.

Only clang based toolchain is currently supported. It has to be activated by setting IDF_TOOLCHAIN=clang in the environment or in CMake cache before configuring the project.

**Prerequisites**

If you have never run this tool before, take the following steps to get this tool prepared.

1. Run `idf_tools.py install esp-clang` to install the clang-tidy required binaries

   **Note:** This toolchain is still under development. After the final release, you do not have to install them manually.

2. Run the export scripts (`export.sh`/`export.bat`/...) again to refresh the environment variables.

**Extra Commands**

**clang-check** Run `idf.py clang-check` to re-generate the compilation database and run clang-tidy under your current project folder. The output would be written to `<project_dir>/warnings.txt`.

Run `idf.py clang-check --help` to see the full documentation.

**clang-html-report**

1. Run `pip install codereport` to install the additional dependency.
2. Run `idf.py clang-html-report` to generate an HTML report in folder `<project_dir>/html_report` according to the `warnings.txt`. Please open the `<project_dir>/html_report/index.html` in your browser to check the report.

**Bug Report**

This tool is hosted in espressif/clang-tidy-runner. If you were to face any bugs or have any feature request, please report them via Github issues.

4.27.6 Downloadable IDF Tools

The ESP-IDF build process relies on a number of tools: cross-compiler toolchains, CMake build system, and others. Installing the tools using an OS-specific package manager (e.g., apt, yum, brew, etc.) is the preferred method, when the required version of the tool is available. This recommendation is reflected in the Get Started. For example, on Linux and macOS, it is recommended to install CMake using an OS package manager.

However, some of the tools are specific to ESP-IDF and are not available in OS package repositories. Furthermore, different ESP-IDF versions require different tool versions for proper operation. To solve these two problems, ESP-IDF provides a set of scripts that can download and install the correct tool versions and set up the environment accordingly.
The rest of the document refers to these downloadable tools simply as “tools”. Other kinds of tools used in ESP-IDF are:

- Python scripts bundled with ESP-IDF such as `idf.py`
- Python packages installed from PyPI

The following sections explain the installation method and provide the list of tools installed on each platform.

**Note:** This document is provided for advanced users who need to customize their installation, users who wish to understand the installation process, and ESP-IDF developers.

If you are looking for instructions on how to install the tools, see *Get Started*.

### Tools Metadata File

The list of tools and tool versions required for each platform is located in `tools/tools.json`. The schema of this file is defined by `tools/tools_schema.json`.

This file is used by the `tools/idf_tools.py` script when installing the tools or setting up the environment variables.

### Tools Installation Directory

The `IDF_TOOLS_PATH` environment variable specifies the location where the tools are to be downloaded and installed. If not set, the default location will be `HOME/.espressif` on Linux and macOS, and `%USER_PROFILE%\.espressif` on Windows.

Inside the `IDF_TOOLS_PATH` directory, the tools installation scripts create the following directories and files:

- `dist` — where the archives of the tools are downloaded.
- `tools` — where the tools are extracted. The tools are extracted into subdirectories: `tools/TOOL_NAME/VERSION/`. This arrangement allows different versions of tools to be installed side by side.
- `idf-env.json` — user install options, such as targets and features, are stored in this file. Targets are selected chip targets for which tools are installed and kept up-to-date. Features determine the Python package set which should be installed. These options will be discussed later.
- `python_env` — not related to the tools; virtual Python environments are installed in the sub-directories. Note that the Python environment directory can be placed elsewhere by setting the `IDF_PYTHON_ENV_PATH` environment variable.
- `espdf.constraints.*.txt` — one constraint file for each ESP-IDF release containing Python package version requirements.

### GitHub Assets Mirror

Most of the tools downloaded by the tools installer are GitHub Release Assets, which are files attached to a software release on GitHub.

If GitHub downloads are inaccessible or slow to access, a GitHub assets mirror can be configured.

To use Espressif’s download server, set the environment variable `IDF_GITHUB_ASSETS` to `dl.espressif.com/github_assets`, or `dl.espressif.cn/github_assets` for faster download in China. When the install process is downloading a tool from `github.com`, the URL will be rewritten to use this server instead.

Any mirror server can be used provided the URL matches the `github.com` download URL format. For any GitHub asset URL that the install process downloads, it will replace `https://github.com` with `https://${IDF_GITHUB_ASSETS}`.

**Note:** The Espressif download server currently does not mirror everything from GitHub, but only files attached as Assets to some releases, as well as source archives for some releases.
**idf_tools.py Script**

The tools/idf_tools.py script bundled with ESP-IDF performs several functions:

- **install:** Download the tool into the `$(IDF_TOOLS_PATH)/dist` directory and extract it into `$(IDF_TOOLS_PATH)/tools/TOOL_NAME/VERSION`. The install command accepts the list of tools to install in the `TOOL_NAME` or `TOOL_NAME@VERSION` format. If all is given, all the tools, including required and optional ones, are installed. If no argument or required is given, only the required tools are installed.

- **download:** Similar to install but doesn’t extract the tools. An optional `--platform` argument may be used to download the tools for the specific platform.

- **export:** Lists the environment variables that need to be set to use the installed tools. For most of the tools, setting the PATH environment variable is sufficient, but some tools require extra environment variables. The environment variables can be listed in either shell or key-value formats, which can be set using the `--format` parameter:
  - export optional parameters:
    - `--unset`: Creates a statement that unsets specific global variables and restores the environment to its state before calling `export.{sh/fish}`.
    - `--add_paths_extras`: Adds extra ESP-IDF-related paths of $PATH to `$(IDF_TOOLS_PATH)/esp-idf.json`, which is used to remove global variables when the active ESP-IDF environment is deactivated. For example, while processing the `export.{sh/fish}` script, if new paths are added to the global variable $PATH, this option saves these new paths to the `$(IDF_TOOLS_PATH)/esp-idf.json` file.
  - shell: Produces output suitable for evaluation in the shell. For example, produce the following output on Linux and macOS:
    ```bash
    export PATH="/home/user/.espressif/tools/tool/v1.0.0/bin:$PATH"
    ```

  Produce the following output on Windows:
    ```cmd
    set "PATH=C:\Users\user\.espressif\tools\tool\v1.0.0\bin;%PATH%"
    ```

- **list:** Lists the known versions of the tools, and indicates which ones are installed. The following option is available to customize the output.
  - `--outdated`: Lists only outdated versions of tools installed in IDF_TOOLS_PATH.

- **check:** For each tool, checks whether the tool is available in the system path and in IDF_TOOLS_PATH.

- **install-python-env:** Creates a Python virtual environment in the `$(IDF_TOOLS_PATH)/python_env` directory or directly in the directory set by the IDF_PYTHON_ENV_PATH environment variable, and install the required Python packages there.
  - An optional `--features` argument allows one to specify a comma-separated list of features to be added or removed.
    1. A feature that begins with `–` will be removed, and features with `+` or without any sign will be added. Example syntax for removing feature XY is `--features=-XY`, and for adding feature XY is `--features=+XY` or `--features=XY`. If both removing and adding options are provided with the same feature, no operation is performed.
    2. For each feature, a requirements file must exist. For example, feature XY is a valid feature if...
${IDF_PATH}/tools/requirements/requirements.XY.txt is an existing file with a list of Python packages to be installed.

3. There is one mandatory core feature ensuring the core functionality of ESP-IDF, e.g., build, flash, monitor, debug in console. There can be an arbitrary number of optional features.

4. The selected list of features is stored in idf-env.json.

5. The requirement files contain a list of the desired Python packages to be installed and the espidf.constraints.*.txt file downloaded from https://dl.espressif.com and stored in ${IDF_TOOLS_PATH}, which contains the package version requirements for a given ESP-IDF version.

Note: Although it is not recommended, the download and use of constraint files can be disabled with the --no-constraints argument or setting the IDF_PYTHON_CHECK_CONSTRAINTS environment variable to no.

- check-python-dependencies: Checks if all required Python packages are installed. Packages from ${IDF_PATH}/tools/requirements/requirements.*.txt files selected by the feature list of idf-env.json are checked with the package versions specified in the espidf.constraints.*.txt file.

Note: The constraint file is downloaded with the install-python-env command. Similar to the install-python-env command, the use of constraint files can be disabled with the --no-constraints argument or setting the IDF_PYTHON_CHECK_CONSTRAINTS environment variable to no.

- uninstall: Prints and removes tools that are currently not used by the active ESP-IDF version.
  - --dry-run: Prints installed unused tools.
  - --remove-archives: Additionally removes all older versions of previously downloaded installation packages.

Install Scripts

Shell-specific user-facing installation scripts are provided in the root directory of ESP-IDF repository to facilitate tools installation. These are:

- install.bat for Windows Command Prompt
- install.ps1 for Powershell
- install.sh for Bash
- install.fish for Fish

Apart from downloading and installing the tools in IDF_TOOLS_PATH, these scripts prepare a Python virtual environment, and install the required packages into that environment.

These scripts accept optionally a comma-separated list of chip targets and --enable-* arguments for enabling features. These arguments are passed to the idf_tools.py script which stores them in idf-env.json. Therefore, chip targets and features can be enabled incrementally.

To install tools for all chip targets, run the scripts without any optional arguments using idf_tools.py install --targets=all. Similarly, to install Python packages for core ESP-IDF functionality, run idf_tools.py install-python-env --features=core.

It is also possible to install tools for specific chip targets. For example, install.sh esp32 installs tools only for ESP32. See Step 3. Set up the Tools for more examples.

install.sh --enable-XY enables feature XY (by running idf_tools.py install-python-env --features=core,XY).

Export Scripts

Since the installed tools are not permanently added to the user or system PATH environment variable, an extra step is required to use them in the command line. The following scripts modify the environment variables in the current shell to make the correct versions of the tools available:
Chapter 4. API Guides

- export.bat for Windows Command Prompt
- export.ps1 for Powershell
- export.sh for Bash
- export.fish for Fish

Note: To modify the shell environment in Bash, export.sh must be “sourced” by using the . ./export.sh command. Please ensure to include the leading dot and space.

export.sh may be used with shells other than Bash (such as zsh). However, in this case, it is required to set the IDF_PATH environment variable before running the script. When used in Bash, the script guesses the IDF_PATH value from its own location.

In addition to calling idf_tools.py, these scripts list the directories that have been added to the PATH.

Other Installation Methods

Depending on the environment, more user-friendly wrappers for idf_tools.py are provided:

- ESP-IDF Tools Installer can download and install the tools. Internally the installer uses idf_tools.py.
- ESP-IDF Eclipse Plugin includes a menu item to set up the tools. Internally the plugin calls idf_tools.py.
- VSCode ESP-IDF Extension includes an onboarding flow. This flow helps set up the tools. Although the extension does not rely on idf_tools.py, the same installation method is used.

Custom Installation

Although the methods above are recommended for ESP-IDF users, they are not a must for building ESP-IDF applications. ESP-IDF build system expects that all the necessary tools are installed somewhere, and made available in the PATH.

Uninstall ESP-IDF

Uninstalling ESP-IDF requires removing both the tools and the environment variables that have been configured during the installation.

- Windows users using the Windows ESP-IDF Tools Installer can simply run the uninstall wizard to remove ESP-IDF.
- To remove an installation performed by running the supported install scripts, simply delete the tools installation directory including the downloaded and installed tools. Any environment variables set by the export scripts are not permanent and will not be present after opening a new environment.
- When dealing with a custom installation, in addition to deleting the tools as mentioned above, you may also need to manually revert any changes to environment variables or system paths that were made to accommodate the ESP-IDF tools (e.g., IDF_PYTHON_ENV_PATH or IDFTOOLS_PATH). If you manually copied any tools, you would need to track and delete those files manually.
- If you installed any plugins like the ESP-IDF Eclipse Plugin or VSCode ESP-IDF Extension, you should follow the specific uninstallation instructions described in the documentation of those components.

Note: Uninstalling the ESP-IDF tools does not remove any project files or your code. Be mindful of what you are deleting to avoid losing any work. If you are unsure about a step, refer back to the installation instructions.

These instructions assume that the tools were installed following the procedures in this provided document. If you’ve used a custom installation method, you might need to adapt these instructions accordingly.
## List of ESP-IDF Tools

### xtensa-esp-elf-gdb  
GDB for Xtensa

**License:** GPL-3.0-or-later  
**More info:** [https://github.com/espressif/binutils-gdb](https://github.com/espressif/binutils-gdb)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
</table>
| linux-amd64| required | https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-x86_64-linux-gnu.tar.gz  
SHA256: d0743ec43cd92c35452a9097f7863281de4e72f04120d63cfbcf9d591a373529 |
SHA256: bc1fac0366ca08e26c45896ca21c8c90efc2cdd431b8ba084e8772e15502d0e |
SHA256: 25efc51d52b71f097ccce763c5c885c8f5026b432fe4b5badd6a5f36fe34d04 |
SHA256: 0f9f39fdec4d8c9c1ef33149a3fcd2cf1baed121529c507817e994d5ac38ca4 |
SHA256: e0af0b3b4a6b29a43cd5f47e331a966d9258f7d2b5b656c6251490f7105b2 |
SHA256: bd146fd99a52b2d71c7ce0f62b9e18f3423d56ca7b2b2c954046b0d7d23142f |
SHA256: 5edc7656bf9d2f2ad24e443d3df7567354f336a65d4af5b2ee805d5ce5c4c24 |
SHA256: ea4f3ee3b695ad1ad2e07108a21a50037a3e4a420cde34b2ba95d612faed989 |
| win64      | required | https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/xtensa-esp-elf-gdb-12.1_20231023-x86_64-w64-mingw32.zip  
SHA256: 13bb97f39173948d1cfb6651d9b335ea9d521f1dd0d1eda3a2d23d8c6344 |

### riscv32-esp-elf-gdb  
GDB for RISC-V

**License:** GPL-3.0-or-later  
**More info:** [https://github.com/espressif/binutils-gdb](https://github.com/espressif/binutils-gdb)
<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-amd64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-linux-gnu.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-linux-gnu.tar.gz</a>&lt;br&gt;SHA256: 2c78b806be176b1e449e07f83429d38dfc39a13f89a127ac1f6c1230537a0</td>
</tr>
<tr>
<td>linux-arm64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-aarch64-linux-gnu.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-aarch64-linux-gnu.tar.gz</a>&lt;br&gt;SHA256: 33f80117c8777aff9f179e27953e41764c5c46b3c576dc96a37ecc7a368807eee</td>
</tr>
<tr>
<td>linux-armel</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-arm-linux-gnueabi.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-arm-linux-gnueabi.tar.gz</a>&lt;br&gt;SHA256: 292e6ec0a9381c4ab0badd15c5aae2586428668fb5d030c9be7deda5e7f707e05</td>
</tr>
<tr>
<td>linux-armhf</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-arm-linux-gnueabihf.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-arm-linux-gnueabihf.tar.gz</a>&lt;br&gt;SHA256: 3b803ab1ae619d62a885af3dc1c2979de77368d59b888c27ec6e525709e78e5f2</td>
</tr>
<tr>
<td>linux-i686</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-i586-linux-gnu.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-i586-linux-gnu.tar.gz</a>&lt;br&gt;SHA256: 68a22fbc6c6371ec4d5e503ec92211977eb2006f0c29e67dbce6b93c77c6b7ec</td>
</tr>
<tr>
<td>macos</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-apple-darwin14.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-apple-darwin14.tar.gz</a>&lt;br&gt;SHA256: 322c7226c612225ed8c9d7f95a0375105756dc5113d369958ce0858ada1a90257</td>
</tr>
<tr>
<td>macos-arm64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-aarch64-apple-darwin21.1.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-aarch64-apple-darwin21.1.tar.gz</a>&lt;br&gt;SHA256: c2224b3a84d02451c530cf004c29653292d963a1b0421b4b472b862b6d6db97e0b</td>
</tr>
<tr>
<td>win32</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-i686-w64-mingw32.zip">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-i686-w64-mingw32.zip</a>&lt;br&gt;SHA256: 4b424149a999d3d87ce7eb8d25e99b9ad966c7f7624253fa8f7771593d7ef69f5602</td>
</tr>
<tr>
<td>win64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-w64-mingw32.zip">https://github.com/espressif/binutils-gdb/releases/download/esp-gdb-v12.1_20231023/riscv32-esp-elf-gdb-12.1_20231023-x86_64-w64-mingw32.zip</a>&lt;br&gt;SHA256: 728231546ad5006d34463f972658b2a89e52f660a42abab08a29be6d4a8046ad</td>
</tr>
</tbody>
</table>

**xtensa-esp-elf**  Toolchain for 32-bit Xtensa based on GCC  
License: GPL-3.0-with-GCC-exception  
More info: https://github.com/espressif/crosstool-NG
<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
</table>
| linux-amd64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/xtensa-esp-elf-13.2.0_20230928-x86_64-linux-gnu.tar.xz
SHA256: baed7a23ea8516f7be42640f4420e4dd1ebf6d64189a14fc330d73e173b3a038b |
| linux-arm64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/xtensa-esp-elf-13.2.0_20230928-aarch64-linux-gnu.tar.xz
SHA256: fa64475bedafbc1e10f9ae01c6c10803ee9ace088b267d7b90a5ee0816c20f9e |
SHA256: 38702870453b8d226f3dc248e2345aee3afa89892da6a6ef866e05a |
| linux-armhf | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/xtensa-esp-elf-13.2.0_20230928-arm-linux-gnueabihf.tar.xz
SHA256: ae8872fe0f7f342ed1a2e202dad15e1fa255aec852e88bbff2725389ddde01 |
SHA256: f2c25701749f365af5f270221e0e8439ce7f2c266eac145af91c6f2f1001de2d6 |
| macos | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/xtensa-esp-elf-13.2.0_20230928-x86_64-apple-darwin.tar.xz
SHA256: b9b7a6d1dc4eb065bf676f9a04729c1c80d6d0b16e4bf6b001252c292951 |
| macos-arm64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/xtensa-esp-elf-13.2.0_20230928-x86_64-apple-darwin.tar.xz
SHA256: b9b7a6d1dc4eb065bf676f9a04729c1c80d6d0b16e4bf6b001252c292951 |
SHA256: 7a2822ef554f3b88b4f5b03770f8632f81e43f2d6c01e9 |
| win64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/xtensa-esp-elf-13.2.0_20230928-x86_64-w64-mingw32.zip
SHA256: 80e3271b7c9b6494ba8494b90054ada2e4e3282f6e5f625d08808372fa64 |

**esp-clang**  Toolchain for all Espressif chips based on clang

License: Apache-2.0

More info: https://github.com/espressif/llvm-project

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
</table>
| linux-amd64 | optional | https://github.com/espressif/llvm-project/releases/download/esp-16.0.0-20230516/llvm-esp-16.0.0-20230516-linux-amd64.tar.xz
SHA256: 3dbd8dd290913a93e8941da8513c451edc49f97982c72d74bb963ef5c45c41e37f |
| linux-arm64 | optional | https://github.com/espressif/llvm-project/releases/download/esp-16.0.0-20230516/llvm-esp-16.0.0-20230516-linux-arm64.tar.xz
SHA256: 687243e5cbefb7cf05603effbdd6fde5769f94d4aff7e519f5b36f1f43c40e6f6 |
SHA256: 935082bb0704420c5ca42b35038bba7802135348a50cac454ae2fb55a0f84c32 |
| macos | optional | https://github.com/espressif/llvm-project/releases/download/esp-16.0.0-20230516/llvm-esp-16.0.0-20230516-macos.tar.xz
SHA256: d9824ac754ab175bce5b7c2b10a6d290a6ec2211b5ed8970f164e2744714a |
| macos-arm64 | optional | https://github.com/espressif/llvm-project/releases/download/esp-16.0.0-20230516/llvm-esp-16.0.0-20230516-macos-arm64.tar.xz
SHA256: ed5621396dc3e48413e14e886ca468e2149a26b855ba4876ab3d3716d7 |
| win64 | optional | https://github.com/espressif/llvm-project/releases/download/esp-16.0.0-20230516/llvm-esp-16.0.0-20230516-win64.tar.xz
SHA256: 598c8241c8bf10f01be88bd21845307cfc404e127041b4ba4e828350a88692883 |
riscv32-esp-elf  Toolchain for 32-bit RISC-V based on GCC
License: GPL-3.0-with-GCC-exception
More info: https://github.com/espressif/crosstool-NG

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
</table>
| linux-amd64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/riscv32-esp-elf-13.2.0_20230928-x86_64-linux-gnu.tar.xz
SHA256: 782feefe354500c5f968e8c91959651be3b9bd7ae8a17af3f3e2b1bf1caad89 |
| linux-arm64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/riscv32-esp-elf-13.2.0_20230928-aarch64-linux-gnu.tar.xz
SHA256: 66e4b30df8b1d8a9ada79396c812b6d99a565746a7d5040c76e2566b3 |
SHA256: 3231ca04eaa4f53dc602ae1cc728151a16c5d424063ac69542b8bf6cde10e7755 |
| linux-armhf | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/riscv32-esp-elf-13.2.0_20230928-arm-linux-gnueabifl.tar.xz
SHA256: eb43ac9dcaad8fe79bdf48d29c4751d41cbcb1fadd831f2779a84f4fb1c5ca0 |
SHA256: 51421bd181392472f8e8242d53da6305a67b21e107f019f69d215987da9684 |
| macos     | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/riscv32-esp-elf-13.2.0_20230928-x86_64-apple-darwin.tar.xz
SHA256: c40c75a1ae0e4b9864adeff321aaa7b57f74eb4b7d011f1252fd6932bb90f |
| macos-arm64 | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/riscv32-esp-elf-13.2.0_20230928-aarch64-apple-darwin.tar.xz
SHA256: c2f9893707c101e3f890a7f7ef5706f0f68f7c7ca1df0f6445894c83f919624f |
SHA256: 37737463826486c9c11e74a140b1b50195dc8686e547c8ee557950ce811741197c |
| win64     | required | https://github.com/espressif/crosstool-NG/releases/download/esp-13.2.0_20230928/riscv32-esp-elf-13.2.0_20230928-x86_64-w64-mingw32.zip
SHA256: 1300a543053d964fa9104482737152e669fd880efc1d54057378d9e6910a1e |

esp32ulp-elf Toolchain for ESP32 ULP coprocessor
License: GPL-3.0-or-later
More info: https://github.com/espressif/binutils-gdb

Espressif Systems 2015
Submit Document Feedback
Release master
<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
<th>SHA256</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-amd64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-amd64.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-amd64.tar.gz</a></td>
<td>SHA256: b1f7801c3a16162e72393ebb772c0cbf64d22d907be7c2c2dac168736e9195fd</td>
</tr>
<tr>
<td>linux-arm64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-arm64.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-arm64.tar.gz</a></td>
<td>SHA256: d6f74b31eb6b31bb13aee256bb7d60f15484cb8b961d6bf67a62867e5563ea5</td>
</tr>
<tr>
<td>linux-armel</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-armel.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-armel.tar.gz</a></td>
<td>SHA256: e107e7a9cd50d630b034f435a16a52db5a57388dc639a99c4e393c5e429711e9</td>
</tr>
<tr>
<td>linux-armhf</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-armhf.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-armhf.tar.gz</a></td>
<td>SHA256: 6c6dd25477b2e758d4669da3774bf664df0f12442c880f17d6df0339e9c3ade9</td>
</tr>
<tr>
<td>linux-i686</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-i686.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-linux-i686.tar.gz</a></td>
<td>SHA256: beb9b6737c975369b6959007739c88f44eb5afbb220f40737071540b2ca1a9064</td>
</tr>
<tr>
<td>macOS</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-macos.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-macos.tar.gz</a></td>
<td>SHA256: 5a952087b621ced16af1e375f6ac1371a61cb51ab7e7b44cbefb5afda2d573de</td>
</tr>
<tr>
<td>macOS-arm64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-macos-arm64.tar.gz">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-macos-arm64.tar.gz</a></td>
<td>SHA256: 73bda8476ef92d4f4abeef96519abbab40e5e3263e8427469447b83cc7bb9b42</td>
</tr>
<tr>
<td>win32</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-win32.zip">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-win32.zip</a></td>
<td>SHA256: 77e44715e7d7a7a9fd0b27653f880efaf3bc1ac843f1492d8a03ed5d91f731</td>
</tr>
<tr>
<td>win64</td>
<td>required</td>
<td><a href="https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-win64.zip">https://github.com/espressif/binutils-gdb/releases/download/esp32ulp-elf-v2.35_20220830/esp32ulp-elf-2.35_20220830-win64.zip</a></td>
<td>SHA256: 525e5b48299869a3fd8db51baad76612c5104bd96952ae6460ad7e5b5a4e21</td>
</tr>
</tbody>
</table>

**cmake**  
CMake build system

On Linux and macOS, it is recommended to install CMake using the OS-specific package manager (like apt, yum, brew, etc.). However, for convenience it is possible to install CMake using idf_tools.py along with the other tools.

License: BSD-3-Clause

More info: https://github.com/Kitware/CMake
## Platform Required Download

<table>
<thead>
<tr>
<th>Platform</th>
<th>Requisite</th>
<th>Download</th>
</tr>
</thead>
</table>
| linux-amd64       | optional  | https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-linux-x86_64.tar.gz  
SHA256: 726f38e659852391e4bce9b059dc20b851aa77f97e4cc5573fe42775a5c16f |
| linux-arm64       | optional  | https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-linux-aarch64.tar.gz  
SHA256: 50c3b8eb9d3a3cde850dd1ea5e46bc5e74dc6d223eeefc1979189651 |
SHA256: 7dc787ef968d4ef92491a4f919b7839ff708a69608b8111c7a737b52481beb0 |
SHA256: 7dc787ef968d4ef92491a4f919b7839ff708a69608b8111c7a737b52481beb0 |
SHA256: 3e0cca74a56d9027dabb845a5a26e42ef8e8b33eb16556da724187a345145e4 |
SHA256: 3e0cca74a56d9027dabb845a5a26e42ef8e8b33eb16556da724187a345145e4 |
| win32             | required  | https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-windows-x86_64.zip  
SHA256: b1ad8c2dbf0778e36fccc9fd61cd4a962e5c1af40aaddbece3d5074bcf2e103c |
| win64             | required  | https://github.com/Kitware/CMake/releases/download/v3.24.0/cmake-3.24.0-windows-x86_64.zip  
SHA256: b1ad8c2dbf0778e36fccc9fd61cd4a962e5c1af40aaddbece3d5074bcf2e103c |

openocd-esp32  OpenOCD for ESP32  
License: GPL-2.0-only  
More info: https://github.com/espressif/openocd-esp32

<table>
<thead>
<tr>
<th>Platform</th>
<th>Requisite</th>
<th>Download</th>
</tr>
</thead>
</table>
| linux-amd64       | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: 61e38e0a13a5c1664624ec1c1397d777d6b9554b0d3453d5b1f7794ce38cc4b |
| linux-arm64       | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: b430352b5d03e0943b76c071a5ba49b7ce6454e613904a85d52970be60b81eecc |
| linux-armel       | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: 5df16d8a91f013a5474f6b3b914c655a9d267996a3b503031b335ac04a4f8d15 |
| linux-armhf       | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: 1b1b80a71b77e5c715aa59e994db97c64454e613904a85d52970be60b81eecc |
| macos             | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: 4bf764934f488af18dca20d152dd36b487f03bea124ec25b6b37a5258a0b |
| macos-arm64       | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: 6dce89048f642eb0559a915b6514f90feb2a95afe21b8f40b0eef2b27824816 |
| win32             | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: ac9d522a63b0816f642d2547bd55c031788035ced85c067d8e7c2862cb1bd0d |
| win64             | required  | https://github.com/espressif/openocd-esp32/releases/download/v0.12.0-esp32-20230921.tar.gz  
SHA256: ac9d522a63b0816f642d2547bd55c031788035ced85c067d8e7c2862cb1bd0d |
ninja  Ninja build system

On Linux and macOS, it is recommended to install ninja using the OS-specific package manager (like apt, yum, brew, etc.). However, for convenience it is possible to install ninja using idf_tools.py along with the other tools.

License: Apache-2.0

More info: https://github.com/ninja-build/ninja

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-amd64</td>
<td>optional</td>
<td><a href="https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-linux.zip">https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-linux.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: b901ba96e486dce37719a070ed4ef3f79debf45f4f6c9f88e7ddec69cb3df77</td>
</tr>
<tr>
<td>macos</td>
<td>optional</td>
<td><a href="https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-mac.zip">https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-mac.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: 482eb23c59a3ed4f158029112de172dd96bb0e97549c4b1ca32d8fad11f873e</td>
</tr>
<tr>
<td>macos-arm64</td>
<td>optional</td>
<td><a href="https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-mac.zip">https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-mac.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: 482eb23c59a3ed4f158029112de172dd96bb0e97549c4b1ca32d8fad11f873e</td>
</tr>
<tr>
<td>win64</td>
<td>required</td>
<td><a href="https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-win.zip">https://github.com/ninja-build/ninja/releases/download/v1.11.1/ninja-win.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: 524b34a1a9a55005eaf868d991e090ab8ce07fa109f1820d40e7642e289abc</td>
</tr>
</tbody>
</table>

idf-exe  IDF wrapper tool for Windows

License: Apache-2.0

More info: https://github.com/espressif/idf_py_exe_tool

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>win32</td>
<td>required</td>
<td><a href="https://github.com/espressif/idf_py_exe_tool/releases/download/v1.0.3/idf-exe-v1.0.3.zip">https://github.com/espressif/idf_py_exe_tool/releases/download/v1.0.3/idf-exe-v1.0.3.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: 7c81ef534562354a5402ab6b90a6eb1cc8473a9f4a7b7af93e5bd23b4a2755</td>
</tr>
<tr>
<td>win64</td>
<td>required</td>
<td><a href="https://github.com/espressif/idf_py_exe_tool/releases/download/v1.0.3/idf-exe-v1.0.3.zip">https://github.com/espressif/idf_py_exe_tool/releases/download/v1.0.3/idf-exe-v1.0.3.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: 7c81ef534562354a5402ab6b90a6eb1cc8473a9f4a7b7af93e5bd23b4a2755</td>
</tr>
</tbody>
</table>

ccache  Ccache (compiler cache)

License: GPL-3.0-or-later

More info: https://github.com/ccache/ccache

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>win64</td>
<td>required</td>
<td><a href="https://github.com/ccache/ccache/releases/download/v4.8/ccache-4.8-windows-x86_64.zip">https://github.com/ccache/ccache/releases/download/v4.8/ccache-4.8-windows-x86_64.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: a2b3bab4bb8318ff5c5b3e074dc25636258bc7e4b51261f7d9be8127fd68309</td>
</tr>
</tbody>
</table>

dfu-util  dfu-util (Device Firmware Upgrade Utilities)

License: GPL-2.0-only

More info: http://dfu-util.sourceforge.net/

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>win64</td>
<td>required</td>
<td><a href="https://dl.espressif.com/dl/dfu-util-0.11-win64.zip">https://dl.espressif.com/dl/dfu-util-0.11-win64.zip</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHA256: 652eb94cb1c074e6dcead9e47ad628922aeb198a4d440a346ab32e7a0e9bf64</td>
</tr>
</tbody>
</table>
**esp-rom-elfs**  ESP ROM ELFs

License: Apache-2.0

More info: https://github.com/espressif/esp-rom-elfs

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>required</td>
<td><a href="https://github.com/espressif/esp-rom-elfs/releases/download/20230320/esp-rom-elfs-20230320.tar.gz">https://github.com/espressif/esp-rom-elfs/releases/download/20230320/esp-rom-elfs-20230320.tar.gz</a> SHA256: 24bcc8cb3287175d4a0bf65e04bf7ef592a10f022acffca0d5e87eee05996d4</td>
</tr>
</tbody>
</table>

**qemu-xtensa**  QEMU for Xtensa


License: GPL-2.0-only

More info: [https://github.com/espressif/qemu](https://github.com/espressif/qemu)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-amd64</td>
<td>optional</td>
<td><a href="https://github.com/espressif/qemu/releases/download/esp-develop-8.0.0-20230522/esp-qemu-xtensa-softmmu-develop_8.0.0_20230522-x86_64-linux-gnu.tar.bz2">https://github.com/espressif/qemu/releases/download/esp-develop-8.0.0-20230522/esp-qemu-xtensa-softmmu-develop_8.0.0_20230522-x86_64-linux-gnu.tar.bz2</a> SHA256: a7e5e779fd593cb15f6d197034dc2f0b427ed9165a4743e2febc6f6a47dfcc618</td>
</tr>
</tbody>
</table>

**qemu-riscv32**  QEMU for RISC-V


License: GPL-2.0-only

More info: [https://github.com/espressif/qemu](https://github.com/espressif/qemu)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-amd64</td>
<td>optional</td>
<td><a href="https://github.com/espressif/qemu/releases/download/esp-develop-8.0.0-20230522/esp-qemu-riscv32-softmmu-develop_8.0.0_20230522-x86_64-linux-gnu.tar.bz2">https://github.com/espressif/qemu/releases/download/esp-develop-8.0.0-20230522/esp-qemu-riscv32-softmmu-develop_8.0.0_20230522-x86_64-linux-gnu.tar.bz2</a> SHA256: bc7607720ff3d7e3d39f3e1810b8795f376f4b9cf3783c8f2ed3f7f14ba74717</td>
</tr>
</tbody>
</table>

### 4.28  Unit Testing in ESP32-S2

ESP-IDF provides the following methods to test software.

- Target based tests using a central unit test application which runs on the esp32s2. These tests use the **Unity** unit test framework. They can be integrated into an ESP-IDF component by placing them in the component’s `test` subdirectory. This document mainly introduces this target based tests.
- Linux-host based unit tests in which part of the hardware can be abstracted via mocks. Currently, Linux-host based tests are still under development and only a small fraction of IDF components support them. More information on running IDF applications on the host can be found here: [Running Applications on the Host Machine](#).
4.28.1 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:

```c
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 tests, 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 (i.e., a test component). ESP-IDF uses the Unity test framework located in the `unity` component. Thus, each test component should specify the `unity` component as a component requirement using the `REQUIRES` argument. 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:

```cmake
idf_component_register(
   SRC_DIRS ".",
   INCLUDE_DIRS ".",
   REQUIRES unity)
```

See [http://www.throwtheswitch.org/unity](http://www.throwtheswitch.org/unity) for more information about writing tests in Unity.

4.28.2 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 through 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:

```c
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,
   }
   // More test cases...
}
```

(continues on next page)
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.

### 4.28.3 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 want to test if the reset reason is correct after a wake up 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()
{
    soc_reset_reason_t reason = esp_rom_get_reset_reason(0);
    TEST_ASSERT(reason == RESET_REASON_CORE_DEEP_SLEEP);
}
```

```
TEST_CASE_MULTIPLE_STAGES("reset reason check for deepsleep", "[esp32s2]", 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.
4.28.4 Tests For Different Targets

Some tests (especially those related to hardware) cannot run on all targets. Below is a guide how to make your unit tests run on only specified targets.

1. Wrap your test code by `!(TEMPORARY_)DISABLED_FOR TARGETS()` macros and place them either in the original test file, or separate the code into files grouped by functions, but make sure all these files will be processed by the compiler. E.g.:

   ```
   #if !TEMPORARY_DISABLED_FOR_TARGETS(ESP32, ESP8266)
   TEST_CASE("a test that is not ready for esp32 and esp8266 yet", "[]")
   {
   }
   #endif
   #if TEMPORARY_DISABLED_FOR_TARGETS(ESP32, ESP8266)
   ```

   Once you need one of the tests to be compiled on a specified target, just modify the targets in the disabled list. It’s more encouraged to use some general conception that can be described in `soc_caps.h` to control the disabling of tests. If this is done but some of the tests are not ready yet, use both of them (and remove `!(TEMPORARY_)DISABLED_FOR_TARGETS()`) later. E.g.:

   ```
   #if SOC_SDIO_SLAVE_SUPPORTED
   #if !TEMPORARY_DISABLED_FOR_TARGETS(ESP64)
   TEST_CASE("a sdio slave tests that is not ready for esp64 yet", "[sdio_slave]")
   {
   //available for esp32 now, and will be available for esp64 in the future
   }
   #endif
   ```

2. For test code that you are 100% for sure that will not be supported (e.g., no peripheral at all), use `DISABLED_FOR_TARGETS`; for test code that should be disabled temporarily, or due to lack of runners, etc., use `TEMPORARY_DISABLED_FOR_TARGETS`.

Some old ways of disabling unit tests for targets, that have obvious disadvantages, are deprecated:

- DON’T put the test code under test/target folder and use CMakelists.txt to choose one of the target folders. This is prevented because test code is more likely to be reused than the implementations. If you put something into test/esp32 just to avoid building it on esp32s2, it’s hard to make the code tidy if you want to enable the test again on esp32s3.
- DON’T use `CONFIG_IDF_TARGET_xxx` macros to disable the test items any more. This makes it harder to track disabled tests and enable them again. Also, a black-list style `#if !disabled` is preferred to white-list style `#if CONFIG_IDF_TARGET_xxx`, since you will not silently disable cases when new targets are added in the future. But for test implementations, it’s allowed to use `#if CONFIG_IDF_TARGET_xxx` to pick one of the implementation code.
  - Test item: some items that will be performed on some targets, but skipped on other targets. E.g. There are three test items SD 1-bit, SD 4-bit and SDSPI. For ESP32-S2, which doesn’t have SD host, among the tests only SDSPI is enabled on ESP32-S2.
  - Test implementation: some code will always happen, but in different ways. E.g. There is no SDIO PKT_LEN register on ESP8266. If you want to get the length from the slave as a step in the test process, you can have different implementation code protected by `#if CONFIG_IDF_TARGET_xxx` reading in different ways. But please avoid using `#else` macro. When new target is added, the test case will fail at building stage, so that the maintainer will be aware of this, and choose one of the implementations explicitly.

4.28.5 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).

**Note:** Due to inherent limitations of Windows command prompt, following syntax has to be used in order to build unit-test-app with multiple components: idf.py -T xxx -T yyy build or with escaped quotes: idf.py -T "xxx yyy" build in PowerShell or idf.py -T "ssd1306 hts221" build in Windows command prompt.

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. For more information, see tools/unit-test-app/README.md.

### 4.28.6 Running Unit Tests

After flashing reset the ESP32-S2 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 (SPI2) to host2 (SPI3)" [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" [esp32s2][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 Naming Conventions

- 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:

<table>
<thead>
<tr>
<th>Sub-menu Example</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running gpio master/slave test example...</td>
<td>1) gpio_master_test</td>
</tr>
<tr>
<td></td>
<td>2) gpioSlave_test</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Sub-menu Example</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running reset reason check for deepsleep...</td>
<td>1) trigger_deepsleep</td>
</tr>
<tr>
<td></td>
<td>2) check_deepsleep_reset_reason</td>
</tr>
</tbody>
</table>

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.

### 4.28.7 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 micro benchmarks). There might be some variability in measured time between runs and between different builds. A technique for eliminating some of the variability is to place code and data in instruction or data RAM (IRAM/DRAM), respectively. The CPU can access IRAM and DRAM directly, eliminating the cache out of the equation. However, this might not always be viable as the size of IRAM and DRAM is limited.

The cache compensated timer is an alternative to placing the code/data to be benchmarked in IRAM/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.

```c
// 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();
```
Chapter 4. API Guides

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 \texttt{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 \texttt{ccomp\_timer\_stop} will be invalid.

### 4.28.8 Mocks

\textbf{Note:} Currently, mocking is only possible with some selected components when running on the Linux host. In the future, we plan to make essential components in IDF mock-able. This will also include mocking when running on the ESP32-S2.

One of the biggest problems regarding unit testing on embedded systems are the strong hardware dependencies. Running unit tests directly on the ESP32-S2 can be especially difficult for higher layer components for the following reasons:

- Decreased test reliability due to lower layer components and/or hardware setup.
- Increased difficulty in testing edge cases due to limitations of lower layer components and/or hardware setup
- Increased difficulty in identifying the root cause due to the large number of dependencies influencing the behavior

When testing a particular component, (i.e., the component under test), mocking allows the dependencies of the component under test to be substituted (i.e., mocked) entirely in software. Through mocking, hardware details are emulated and specified at run time, but only if necessary. To allow mocking, ESP-IDF integrates the CMock mocking framework as a component. With the addition of some CMake functions in the ESP-IDF build system, it is possible to conveniently mock the entirety (or a part) of an IDF component.

Ideally, all components that the component under test is dependent on should be mocked, thus allowing the test environment complete control over all interactions with the component under test. However, if mocking all dependent components becomes too complex or too tedious (e.g., because you need to mock too many function calls) you have the following options:

- Include more “real” IDF code in the tests. This may work but increases the dependency on the “real” code’s behavior. Furthermore, once a test fails, you may not know if the failure is in your actual code under test or the “real” IDF code.
- Re-evaluate the design of the code under test and attempt to reduce its dependencies by dividing the code under test into more manageable components. This may seem burdensome but it is quite common that unit tests expose software design weaknesses. Fixing design weaknesses will not only help with unit testing in the short term, but will help future code maintenance as well.

Refer to \texttt{cmock/CMock/docs/CMock\_Summary.md} for more details on how CMock works and how to create and use mocks.

### Requirements

Mocking with CMock requires Ruby on the host machine. Furthermore, since mocking currently only works on the Linux target, the requirements of the latter also need to be fulfilled:

- Installed ESP-IDF including all ESP-IDF requirements
- System package requirements (\texttt{libbsd, libbsd-dev})
- A recent enough Linux or macOS version and GCC compiler
- All components the application depends on must be either supported on the Linux target (Linux/POSIX simulator) or mock-able

An application that runs on the Linux target has to set the \texttt{COMPONENTS} variable to \texttt{main} in the \texttt{CMakeLists.txt} of the application’s root directory:
This prevents the automatic inclusion of all components from ESP-IDF to the build process which is otherwise done for convenience.

**Mock a Component**

If a mocked component, called a *component mock*, is already available in ESP-IDF, then it can be used right away as long as it satisfies the required functionality. Refer to *Component Linux/Mock Support Overview* to see which components are mocked already. Then refer to *Adjustments in Unit Test* in order to use the component mock.

It is necessary to create component mocks if they are not yet provided in ESP-IDF. To create a component mock, the component needs to be overwritten in a particular way. Overriding a component entails creating a component with the exact same name as the original component, then letting the build system discover it later than the original component (see *Multiple components with the same name* for more details).

In the component mock, the following parts are specified:

- The headers providing the functions to generate mocks for
- Include paths of the aforementioned headers
- Dependencies of the mock component (this is necessary e.g. if the headers include files from other components)

All these parts have to be specified using the IDF build system function `idf_component_mock`. You can use the IDF build system function `idf_component_get_property` with the tag `COMPONENT_OVERRIDEN_DIR` to access the component directory of the original component and then register the mock component parts using `idf_component_mock`:

```
idf_component_get_property(original_component_dir <original-component-name>..-
   COMPONENT_OVERRIDEN_DIR)
...
idf_component_mock(INCLUDE_DIRS "${original_component_dir}/include"
   REQUIRES freertos
   MOCK_HEADER_FILES ${original_component_dir}/include/header_containing_-
   <functions_to_mock.h)
```

The component mock also requires a separate mock directory containing a `mock_config.yaml` file that configures CMock. A simple `mock_config.yaml` could look like this:

```
:cmock:
  :plugins:
    - expect
    - expect_any_args
```

For more details about the CMock configuration yaml file, have a look at `cmock/CMock/docs/CMock_Summary.md`. Note that the component mock does not have to mock the original component in its entirety. As long as the test project’s dependencies and dependencies of other code to the original components are satisfied by the component mock, partial mocking is adequate. In fact, most of the component mocks in IDF in `tools/mocks` are only partially mocking the original component.

Examples of component mocks can be found under `tools/mocks` in the IDF directory. General information on how to override an IDF component can be found in *Multiple components with the same name*. There are several examples for testing code while mocking dependencies with CMock (non-exhaustive list):

- unit test for the NVS Page class.
- unit test for esp_event.
- unit test for mqtt.
Adjustments in Unit Test

The unit test needs to inform the cmake build system to mock dependent components (i.e., it needs to override the original component with the mock component). This is done by either placing the component mock into the project’s components directory or adding the mock component’s directory using the following line in the project’s root CMakeLists.txt:

```
list(APPEND EXTRA_COMPONENT_DIRS "<mock_component_dir>")
```

Both methods will override existing components in ESP-IDF with the component mock. The latter is particularly convenient if you use component mocks that are already supplied by IDF.

Users can refer to the esp_event host-based unit test and its esp_event/host_test/esp_event_unit_test/CMakeLists.txt as an example of a component mock.

4.29 Running ESP-IDF Applications on Host

**Note:** Running ESP-IDF applications on host is currently still an experimental feature, thus there is no guarantee for API stability. However, user feedback via the ESP-IDF GitHub repository or the ESP32 forum is highly welcome, and may help influence the future of design of the ESP-IDF host-based applications.

This document provides an overview of the methods to run ESP-IDF applications on Linux, and what type of ESP-IDF applications can typically be run on Linux.

4.29.1 Introduction

Typically, an ESP-IDF application is built (cross-compiled) on a host machine, uploaded (i.e., flashed) to an ESP chip for execution, and monitored by the host machine via a UART/USB port. However, execution of an ESP-IDF application on an ESP chip can be limiting in various development/usage/testing scenarios.

Therefore, it is possible for an ESP-IDF application to be built and executed entirely within the same Linux host machine (henceforth referred to as “running on host”). Running ESP-IDF applications on host has several advantages:

- No need to upload to a target.
- Faster execution on a host machine, compared to running on an ESP chip.
- No requirements for any specific hardware, except the host machine itself.
- Easier automation and setup for software testing.
- Large number of tools for code and runtime analysis, e.g., Valgrind.

A large number of ESP-IDF components depend on chip-specific hardware. These hardware dependencies must be mocked or simulated when running on host. ESP-IDF currently supports the following mocking and simulation approaches:

1. Using the FreeRTOS POSIX/Linux simulator that simulates FreeRTOS scheduling. On top of this simulation, other APIs are also simulated or implemented when running on host.
2. Using CMock to mock all dependencies and run the code in complete isolation.

In principle, it is possible to mix both approaches (POSIX/Linux simulator and mocking using CMock), but this has not been done yet in ESP-IDF. Note that despite the name, the FreeRTOS POSIX/Linux simulator currently also works on macOS. Running ESP-IDF applications on host machines is often used for testing. However, simulating the environment and mocking dependencies does not fully represent the target device. Thus, testing on the target device is still necessary, though with a different focus that usually puts more weight on integration and system testing.
Chapter 4. API Guides

**Note:** Another possibility to run applications on the host is to use the QEMU simulator. However, QEMU development for ESP-IDF applications is still a work in progress and has not been documented yet.

**CMock-Based Approach**

This approach uses the CMock framework to solve the problem of missing hardware and software dependencies. CMock-based applications running on the host machine have the added advantage that they usually only compile the necessary code, i.e., the (mostly mocked) dependencies instead of the entire system. For a general introduction toMocks and how to configure and use them in ESP-IDF, please refer to *Mocks*.

**POSIX/Linux Simulator Approach**

The FreeRTOS POSIX/Linux simulator is available on ESP-IDF as a preview target already. This simulator allows ESP-IDF components to be implemented on the host, making them accessible to ESP-IDF applications when running on host. Currently, only a limited number of components are ready to be built on Linux. Furthermore, the functionality of each component ported to Linux may also be limited or different compared to the functionality when building that component for a chip target. For more information about whether the desired components are supported on Linux, please refer to *Component Linux/Mock Support Overview*.

**Note:** The FreeRTOS POSIX/Linux simulator allows configuring the Amazon SMP FreeRTOS version. However, the simulation still runs in single-core mode. The main reason allowing Amazon SMP FreeRTOS is to provide API compatibility with ESP-IDF applications written for Amazon SMP FreeRTOS.

### 4.29.2 Requirements for Using Mocks

- Installed ESP-IDF including all ESP-IDF requirements
- System package requirements (libbsd, libbsd-dev)
- A recent enough Linux or macOS version and GCC compiler
- All components the application depends on must be either supported on the Linux target (Linux/POSIX simulator) or mock-able

An application that runs on the Linux target has to set the `COMPONENTS` variable to `main` in the `CMakeLists.txt` of the application’s root directory:

```bash
set(COMPONENTS main)
```

This prevents the automatic inclusion of all components from ESP-IDF to the build process which is otherwise done for convenience.

If any mocks are used, then Ruby is required, too.

### 4.29.3 Build and Run

To build the application on Linux, the target has to be set to `linux` and then it can be built and run:

```bash
idf.py --preview set-target linux
didf.py build
didf.py monitor
```
4.29.4 Component Linux/Mock Support Overview

Note that any “Yes” here does not necessarily mean a full implementation or mocking. It can also mean a partial implementation or mocking of functionality. Usually, the implementation or mocking is done to a point where enough functionality is provided to build and run a test application.

<table>
<thead>
<tr>
<th>Component</th>
<th>Mock</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmock</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>driver</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>esp_common</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_event</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_http_client</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_http_server</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_https_server</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_hw_support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_netif</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_netif_stack</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_partition</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_rom</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_system</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>esp_timer</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>esp_tls</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>fatfs</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>freertos</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>hal</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>heap</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>http_parser</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>json</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>linux</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>log</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>lwip</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mbedtls</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>mbdssl</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>mqtt</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>nvs_flash</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>partition_table</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>protobuf-c</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>pthread</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>soc</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>spiffs</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>spi_flash</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>tcp_transport</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>unity</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

4.30 USB OTG Console

On chips with an integrated USB peripheral, it is possible to use USB Communication Device Class (CDC) to implement the serial console, instead of using UART with an external USB-UART bridge chip. ESP32-S2 ROM code contains a USB CDC implementation, which supports for some basic functionality without requiring the application to include the USB stack:

- Bidirectional serial console, which can be used with *IDF Monitor* or another serial monitor
- Flashing using *esptool.py* and *idf.py flash*
- *Device Firmware Update (DFU)* interface for flashing the device using *dfu-util* and *idf.py dfu*.
Note: At the moment, this “USB Console” feature is incompatible with TinyUSB stack. However, if TinyUSB is used, it can provide its own CDC implementation.

4.30.1 Hardware Requirements

Connect ESP32-S2 to the USB port as follows

<table>
<thead>
<tr>
<th>GPIO</th>
<th>USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>D+ (green)</td>
</tr>
<tr>
<td>19</td>
<td>D- (white)</td>
</tr>
<tr>
<td>GND</td>
<td>GND (black)</td>
</tr>
<tr>
<td></td>
<td>+5V (red)</td>
</tr>
</tbody>
</table>

Some development boards may offer a USB connector for the internal USB peripheral—in that case, no extra connections are required.

4.30.2 Software Configuration

USB console feature can be enabled using `CONFIG_ESP_CONSOLE_USB_CDC` option in menuconfig tool (see `CONFIG_ESP_CONSOLE_UART`).

Once the option is enabled, build the project as usual.

4.30.3 Uploading the Application

Initial Upload

If the ESP32-S2 is not yet flashed with a program that enables a USB console, an initial upload of the program is required. There are 3 alternative options to perform the initial upload.

Once the initial upload is done, the application will start up and a USB CDC port will appear in the system.

Note: The port name may change after the initial upload, so check the port list again before running `idf.py monitor`.

Initial Upload Using the ROM Download Mode, over USB CDC

- Place ESP32-S2 into download mode. To do this, keep GPIO0 low while toggling reset. On many development boards, the “Boot” button is connected to GPIO0, and you can press “Reset” button while holding “Boot”.
- A serial port will appear in the system. On most operating systems (Windows 8 and later, Linux, macOS) driver installation is not required. Find the port name using Device Manager (Windows) or by listing `/dev/ttyACM*` devices on Linux or `/dev/cu*` devices on macOS.
- Run `idf.py flash -p PORT` to upload the application, with `PORT` determined in the previous step.

Initial Upload Using the ROM Download Mode, over USB DFU

- Place ESP32-S2 into download mode. To do this, keep GPIO0 low while toggling reset. On many development boards, the “Boot” button is connected to GPIO0, and you can press “Reset” button while holding “Boot”.
- Run `idf.py dfu-flash`.

See *Flashing the DFU Image* for details about DFU flashing.
**Chapter 4. API Guides**

**Initial Upload Using UART**  On development boards with a USB-UART bridge, upload the application over UART: `idf.py flash -p PORT` where `PORT` is the name of the serial port provided by the USB-UART bridge.

**Subsequent Usage**

Once the application is uploaded for the first time, you can run `idf.py flash` and `idf.py monitor` as usual.

### 4.30.4 Limitations

There are several limitations to the USB console feature. These may or may not be significant, depending on the type of application being developed, and the development workflow. Most of these limitations stem from the fact that USB CDC is implemented in software, so the console working over USB CDC is more fragile and complex than a console working over UART.

1. If the application crashes, panic handler output may not be sent over USB CDC in some cases. If the memory used by the CDC driver is corrupted, or there is some other system-level issue, CDC may not work for sending panic handler messages over USB. This does work in many situations, but is not guaranteed to work as reliably as the UART output does. Similarly, if the application enters a boot loop before the USB CDC driver has a chance to start up, there will be no console output.

2. If the application accidentally reconfigures the USB peripheral pins, or disables the USB peripheral, USB CDC device will disappear from the system. After fixing the issue in the application, you will need to follow the Initial Upload process to flash the application again.

3. If the application enters light sleep (including automatic light sleep) or deep sleep mode, USB CDC device will disappear from the system.

4. USB CDC driver reserves some amount of RAM and increases application code size. Keep this in mind if trying to optimize application memory usage.

5. By default, the low-level `esp_rom_printf` feature and `ESP_EARLY_LOG` are disabled when USB CDC is used. These can be enabled using `CONFIG_ESP_CONSOLE_USB_CDC_SUPPORT_ETS_PRINTF` option. With this option enabled, `esp_rom_printf` can be used, at the expense of increased IRAM usage. Keep in mind that the cost of `esp_rom_printf` and `ESP_EARLY_LOG` over USB CDC is significantly higher than over UART. This makes these logging mechanisms much less suitable for “printf debugging”, especially in the interrupt handlers.

6. If you are developing an application which uses the USB peripheral with the TinyUSB stack, this USB Console feature can not be used. This is mainly due to the following reasons:
   - This feature relies on a different USB CDC software stack in ESP32-S2 ROM.
   - USB descriptors used by the ROM CDC stack may be different from the descriptors used by TinyUSB.
   - When developing applications which use USB peripheral, it is very likely that USB functionality will not work or will not fully work at some moments during development. This can be due to misconfigured USB descriptors, errors in the USB stack usage, or other reasons. In this case, using the UART console for flashing and monitoring provides a much better development experience.

7. When debugging the application using JTAG, USB CDC may stop working if the CPU is stopped on a break-point. USB CDC operation relies on interrupts from the USB peripheral being serviced periodically. If the host computer does not receive valid responses from the USB device side for some time, it may decide to disconnect the device. The actual time depends on the OS and the driver, and ranges from a few hundred milliseconds to a few seconds.

### 4.31 Wi-Fi Driver

#### 4.31.1 ESP32-S2 Wi-Fi Feature List

The following features are supported:
4.3.1.2 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 you to select an example.

Setting Wi-Fi Compile-time Options

Refer to Wi-Fi Menuconfig.

Init Wi-Fi

Refer to ESP32-S2 Wi-Fi station General Scenario and ESP32-S2 Wi-Fi AP General Scenario.

Start/Connect Wi-Fi

Refer to ESP32-S2 Wi-Fi station General Scenario and ESP32-S2 Wi-Fi AP General Scenario.

Event-Handling

Generally, it is easy to write code in “sunny-day” scenarios, such as WIFI_EVENT_STA_START and WIFI_EVENT_STA_CONNECTED. The hard part is to write routines in “rainy-day” scenarios, such as WIFI_EVENT_STA_DISCONNECTED. Good handling of “rainy-day” scenarios is fundamental to robust Wi-Fi applications. Refer to ESP32-S2 Wi-Fi Event Description, ESP32-S2 Wi-Fi station General Scenario, and ESP32-S2 Wi-Fi AP General Scenario. See also the overview of the Event Loop Library 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-S2 Wi-Fi API Error Code.
## 4.31.3 ESP32-S2 Wi-Fi API Error Code

All of the ESP32-S2 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`.
- 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 microseconds` 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 and also 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_common/include/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, it is strongly recommended that API users write their own error-handling code.

## 4.31.4 ESP32-S2 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.
- 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 that 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 ESP-IDF is updated to a new release.

## 4.31.5 ESP32-S2 Wi-Fi Programming Model

The ESP32-S2 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, and event task. 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 posts 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.

## 4.31.6 ESP32-S2 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-S2 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 station/AP, 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 station/AP, 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()` 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 or the authentication times out. If there are more than one AP with the same SSID, the disconnected event will be 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, or the AP’s authentication mode is changed.

Upon receiving this event, the default behaviors of the event task are:

- Shutting down the station’s LwIP netif.
- Notifying 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 is the application’s responsibility to distinguish whether the event is caused by `esp_wifi_disconnect()` or other reasons. Sometimes a better reconnection strategy is required. Refer to Wi-Fi Reconnect and Scan When Wi-Fi Is Connecting.

Another thing that deserves attention is that the default behavior of LwIP is to abort all TCP socket connections on receiving the disconnect. In most cases, it is not a problem. However, for some special applications, this may not be what they want. Consider the 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 ignore 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 scenarios, 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 the 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-S2, or when this address changes. The event means that everything is ready and the application can begin its tasks, e.g., creating sockets.

**IP_EVENT_STA_LOST_IP**

This event arises when the IPv4 address becomes invalid.

IP_EVENT_STA_LOST_IP does not arise immediately after the Wi-Fi disconnects. Instead, it starts an IPv4 address lost timer. If the IPv4 address is got before the timer expires, IP_EVENT_STA_LOST_IP does not happen. Otherwise, the event arises when the IPv4 address lost timer expires.

Generally, the application can ignore this event, because it is just a debug event to inform 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-S2 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.

**WIFI_EVENT_AP_STADISCONNECTED**

This event can happen in the following scenarios:

- The application calls `esp_wifi_disconnect()`, or `esp_wifi_deauth_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. 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.

**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.

**WIFI_EVENT_STA_BEACON_TIMEOUT**

If the station does not receive the beacon of the connected AP within the inactive time, the beacon timeout happens, the `WIFI_EVENT_STA_BEACON_TIMEOUT` will arise. The application can set inactive time via API `esp_wifi_set_inactive_time()`.
WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START

The WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START will arise at the start of connectionless module Interval. See connectionless module power save.

4.31.7 ESP32-S2 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_create() 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 initialization 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, the country code, and the bandwidth. Refer to ESP32-S2 Wi-Fi Configuration.

   Generally, the Wi-Fi driver should be configured before the Wi-Fi connection is set up. 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 phases after the WIFI_EVENT_STA_START arises.
Fig. 46: Sample Wi-Fi Event Scenarios in Station Mode
4. Wi-Fi Connect Phase

- s4.1: Once `esp_wifi_connect()` is called, the Wi-Fi driver will start the internal scan/connection process.
- 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.

In step 4.2, the Wi-Fi connection may fail because, for example, the password is wrong, or the AP is not found. 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. 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., the AP is powered off or the RSSI is poor, `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. The recommended actions are: 1) call `esp_wifi_connect()` to reconnect the Wi-Fi, 2) close all sockets, and 3) re-create them if necessary. For details, please 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.

4.31.8 ESP32-S2 Wi-Fi AP General Scenario

Below is a “big scenario” which describes some small scenarios in AP mode:
Fig. 47: Sample Wi-Fi Event Scenarios in AP Mode
4.31.9 ESP32-S2 Wi-Fi Scan

Currently, the `esp_wifi_scan_start()` API is supported only in station or station/AP mode.

### Scan Type

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Scan</td>
<td>Scan by sending a probe request. The default scan is an active scan.</td>
</tr>
<tr>
<td>Passive Scan</td>
<td>No probe request is sent out. Just switch to the specific channel and wait for a beacon. Application can enable it via the scan_type field of <code>wifi_scan_config_t</code>.</td>
</tr>
<tr>
<td>Foreground Scan</td>
<td>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.</td>
</tr>
<tr>
<td>Background Scan</td>
<td>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.</td>
</tr>
<tr>
<td>All-Channel Scan</td>
<td>It scans all of the channels. If the channel field of <code>wifi_scan_config_t</code> is set to 0, it is an all-channel scan.</td>
</tr>
<tr>
<td>Specific Channel Scan</td>
<td>It scans specific channels only. If the channel field of <code>wifi_scan_config_t</code> set to 1-14, it is a specific-channel scan.</td>
</tr>
</tbody>
</table>

The scan modes in above table can be combined arbitrarily, so there are in total 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`. 
There are also some global scan attributes which are configured by API `esp_wifi_set_config()`, refer to `Station Basic Configuration`.

### Scan All APs on 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**

- s2.1: The Wi-Fi driver switches to channel 1. In this case, the scan type is `WIFI_SCAN_TYPE_ACTIVE`, and a probe request is broadcasted. 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 the 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.
Fig. 48: Foreground Scan of all Wi-Fi Channels
Chapter 4. API Guides

• 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 on 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 Specific AP on All Channels

Scenario:
This scan is similar to `Scan All APs on 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.

It is a possible situation that there are multiple APs that match the target AP info, e.g., two APs with the SSID of “ap” are scanned. In this case, 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, the Wi-Fi driver will try to scan the configured AP first. The scan in “Wi-Fi Connect” is the same as `Scan for Specific AP On All Channels`, except that no scan-done event will be generated when the scan is completed. If the target AP is found, the Wi-Fi driver will start the Wi-Fi connection; otherwise, `WIFI_EVENT_STA_DISCONNECTED` will be generated. Refer to `Scan for Specific AP On All 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-S2’s Wi-Fi functionality improves continuously.
Fig. 49: Background Scan of all Wi-Fi Channels
Fig. 50: Scan of specific Wi-Fi Channels
Scan When Wi-Fi Is Connecting

The `esp_wifi_scan_start()` fails immediately if the Wi-Fi is connecting, because the connecting has higher priority than the scan. If scan fails because of connecting, the recommended strategy is to delay for some time 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 the following scenarios:

- The station is connecting a non-existing AP or it connects the existing AP with a wrong password, it always raises the event `WIFI_EVENT_STA_DISCONNECTED`.
- The application calls `esp_wifi_connect()` to reconnect on receiving the disconnect event.
- Another application task, e.g., the console task, calls `esp_wifi_scan_start()` to do scan, the scan always fails immediately because the station keeps connecting.
- When scan fails, the application simply delays for some time and retries the scan.

In the above scenarios, the scan will never succeed because the connecting is in process. So if the application supports similar scenario, it needs to implement a better reconnection strategy. For example:

- The application can choose to define a maximum continuous reconnection counter and stop reconnecting once the counter reaches the maximum.
- The application can choose to reconnect immediately in the first N continuous reconnection, then give a delay sometime and reconnect again.

The application can define its own reconnection strategy to avoid the scan starve to death. Refer to `<Wi-Fi Reconnect>`.

4.31.10 ESP32-S2 Wi-Fi Station Connecting Scenario

This scenario depicts the case if only one target AP is found in the scan phase. For scenarios where more than one AP with the same SSID is found, refer to ESP32-S2 Wi-Fi Station Connecting When Multiple APs Are Found.

Generally, the application can ignore 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.
- 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.
Fig. 5.1: Wi-Fi Station Connecting Process
• s3.3: The association response is received and the association timer is stopped.
• s3.4: The AP rejects the association in the response and the reason code 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 handshake timer is enabled, and the 1/4 EAPOL is not received before the handshake timer expires. WIFI_EVENT_STA_DISCONNECTED will arise and the reason code will be WIFI_REASON_HANDSHAKE_TIMEOUT. Refer to Wi-Fi Reason Code.
• s4.2: The 1/4 EAPOL is received.
• s4.3: The station replies 2/4 EAPOL.
• s4.4: If the 3/4 EAPOL is not received before the handshake timer expires, WIFI_EVENT_STA_DISCONNECTED will arise and the reason code will be WIFI_REASON_HANDSHAKE_TIMEOUT. Refer to Wi-Fi Reason Code.
• s4.5: The 3/4 EAPOL is received.
• s4.6: The station replies 4/4 EAPOL.
• s4.7: The station raises WIFI_EVENT_STA_CONNECTED.

Wi-Fi Reason Code

The table below shows the reason-code defined in ESP32-S2. The first column is the macro name defined in esp_wifi/include/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 9.4.1.7 of IEEE 802.11-2020. (For more information, refer to the standard mentioned above.) The last column describes the reason.

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Mapped To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSPECIFIED</td>
<td>1</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>AUTH_EXPIRE</td>
<td>2</td>
<td></td>
<td>The previous authentication is no longer valid. For the ESP station, this reason is reported when: • auth is timed out. • the reason is received from the AP. For the ESP AP, this reason is reported when: • the AP has not received any packets from the station in the past five minutes. • the AP is stopped by calling esp_wifi_stop(). • the station is de-authed by calling esp_wifi_deauth_sta().</td>
</tr>
<tr>
<td>AUTH_BEAVER</td>
<td>3</td>
<td></td>
<td>De-authenticated, because the sending station is leaving (or has left). For the ESP station, this reason is reported when: • it is received from the AP.</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Mapped To</th>
<th>Description</th>
</tr>
</thead>
</table>
| AS-SOC_EXPIRE | 4     | 4         | Disassociated due to inactivity. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP has not received any packets from the station in the past five minutes. • the AP is stopped by calling `esp_wifi_stop()`. • the station is de-authed by calling `esp_wifi_deauth_sta()`.
| AS-SOC_TOO_MANY | 5     | 5         | Disassociated, because the AP is unable to handle all currently associated STAs at the same time. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the stations associated with the AP reach the maximum number that the AP can support. |
| NOT_AUTHED   | 6     | 6         | Class-2 frame received from a non-authenticated STA. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP receives a packet with data from a non-authenticated station. |
| NOT_ASSOCED  | 7     | 7         | Class-3 frame received from a non-associated STA. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP receives a packet with data from a non-associated station. |
| AS-SOC_LEAVE | 8     | 8         | Disassociated, because the sending station is leaving (or has left) BSS. For the ESP station, this reason is reported when: • it is received from the AP. • the station is disconnected by `esp_wifi_disconnect()` and other APIs. |
| AS-SOC_NOT_AUXED | 9    | 9         | station requesting (re)association is not authenticated by the responding STA. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • the AP receives packets with data from an associated, yet not authenticated, station. |
| DIS-AS-SOC_PWRCAP_BAD | 10   | 10        | Disassociated, because the information in the Power Capability element is unacceptable. For the ESP station, this reason is reported when: • it is received from the AP. |

continues on next page
<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Mapped To</th>
<th>Description</th>
</tr>
</thead>
</table>
| DIS-ASSOC_SUPCHAN_BAD    | 11    | 11        | Disassociated, because the information in the Supported Channels element is unacceptable. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
| IE_INVALID               | 13    | 13        | Invalid element, i.e., an element whose content does not meet the specifications of the Standard in frame formats clause. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
|                          |       |           | For the ESP AP, this reason is reported when:  
|                          |       |           | • the AP parses a wrong WPA or RSN IE.                                                                                                      |
| MIC_FAILURE              | 14    | 14        | Message integrity code (MIC) failure. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
| 4WAY_HANDSHAKE_TIMEOUT   | 15    | 15        | Four-way handshake times out. For legacy reasons, in ESP this reason code is replaced with WIFI_REASON_HANDSHAKE_TIMEOUT. For the ESP station, this reason is reported when:  
|                          |       |           | • the handshake times out.                                                                                                                  |
|                          |       |           | • it is received from the AP.                                                                                                               |
| GROUP_KEY_UPDATE_TIMEOUT | 16    | 16        | Group-Key Handshake times out. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
| IE_IN_4WAY_DIFFERS       | 17    | 17        | The element in the four-way handshake is different from the (Re-)Association Request/Probe and Response/Beacon frame. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
|                          |       |           | • the station finds that the four-way handshake IE differs from the IE in the (Re-)Association Request/Probe and Response/Beacon frame.        |
| GROUP_CIPHER_INVALID     | 18    | 18        | Invalid group cipher. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
| PAIRWISE_CIPHER_INVALID  | 19    | 19        | Invalid pairwise cipher. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
| AKMP_INVALID             | 20    | 20        | Invalid AKMP. For the ESP station, this reason is reported when:  
|                          |       |           | • it is received from the AP.                                                                                                               |
| UNSUPPORTED_IE_VERSION   | 21    | 21        | Unsupported RSNE version. For the ESP station, this reason is reported when:  
<p>|                          |       |           | • it is received from the AP.                                                                                                               |</p>
<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Mapped To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN-VALID_RSN_IE_CAP</td>
<td>22</td>
<td>22</td>
<td>Invalid RSNE capabilities. For the ESP station, this reason is reported when: • it is received from the AP.</td>
</tr>
<tr>
<td>802_1X_AUTH_FAILED</td>
<td>23</td>
<td>23</td>
<td>IEEE 802.1X authentication failed. For the ESP station, this reason is reported when: • it is received from the AP. For the ESP AP, this reason is reported when: • IEEE 802.1X authentication fails.</td>
</tr>
<tr>
<td>CIPHER_SUITE_REJECTED</td>
<td>24</td>
<td>24</td>
<td>Cipher suite rejected due to security policies. For the ESP station, this reason is reported when: • it is received from the AP.</td>
</tr>
<tr>
<td>TDLS_PEER_UNREACHABLE</td>
<td>25</td>
<td>25</td>
<td>TDLS direct-link teardown due to TDLS peer STA unreachable via the TDLS direct link.</td>
</tr>
<tr>
<td>TDLS_UNSPECIFIED</td>
<td>26</td>
<td>26</td>
<td>TDLS direct-link teardown for unspecified reason.</td>
</tr>
<tr>
<td>SSP_REQUESTED_DISASSOC</td>
<td></td>
<td></td>
<td>Disassociated because session terminated by SSP request.</td>
</tr>
<tr>
<td>NO_SSP_ROAMING_AGREEMENT</td>
<td></td>
<td></td>
<td>Disassociated because of lack of SSP roaming agreement.</td>
</tr>
<tr>
<td>BAD_CIPHER_OR_AKM</td>
<td>27</td>
<td>27</td>
<td>Requested service rejected because of SSP cipher suite or AKM requirement.</td>
</tr>
<tr>
<td>NOT_AUTHORIZED_THIS_LOCATION</td>
<td></td>
<td></td>
<td>Requested service not authorized in this location.</td>
</tr>
<tr>
<td>SERVICE_CHANGE_PRECLUDES_TS</td>
<td>31</td>
<td>31</td>
<td>TS deleted because QoS AP lacks sufficient bandwidth for this QoS STA due to a change in BSS service characteristics or operational mode (e.g., an HT BSS change from 40 MHz channel to 20 MHz channel).</td>
</tr>
<tr>
<td>UNSPECIFIED_QOS</td>
<td>32</td>
<td>32</td>
<td>Disassociated for unspecified, QoS-related reason.</td>
</tr>
<tr>
<td>NOT_ENOUGH_BANDWIDTH</td>
<td></td>
<td></td>
<td>Disassociated because QoS AP lacks sufficient bandwidth for this QoS STA.</td>
</tr>
<tr>
<td>MISSING_ACKS</td>
<td>34</td>
<td>34</td>
<td>Disassociated because excessive number of frames need to be acknowledged, but are not acknowledged due to AP transmissions and/or poor channel conditions.</td>
</tr>
<tr>
<td>EXCEEDED_TXOP</td>
<td>35</td>
<td>35</td>
<td>Disassociated because STA is transmitting outside the limits of its TXOPs.</td>
</tr>
<tr>
<td>STA_LEAVING</td>
<td>36</td>
<td>36</td>
<td>Requesting STA is leaving the BSS (or resetting).</td>
</tr>
<tr>
<td>END_BA</td>
<td>37</td>
<td>37</td>
<td>Requesting STA is no longer using the stream or session.</td>
</tr>
<tr>
<td>UNKNOWN_BA</td>
<td>38</td>
<td>38</td>
<td>Requesting STA received frames using a mechanism for which a setup has not been completed.</td>
</tr>
<tr>
<td>TIME_OUT</td>
<td>39</td>
<td>39</td>
<td>Requested from peer STA due to timeout</td>
</tr>
<tr>
<td>Reserved</td>
<td>40 ~ 45</td>
<td>40 ~ 45</td>
<td></td>
</tr>
<tr>
<td>PEER_INITIATED</td>
<td>46</td>
<td></td>
<td>In a Disassociation frame: Disassociated because authorized access limit reached.</td>
</tr>
<tr>
<td>AP_INITIATED</td>
<td>47</td>
<td></td>
<td>In a Disassociation frame: Disassociated due to external service requirements.</td>
</tr>
<tr>
<td>IN-VALID_FT_ACTION_FRAME_COUNT</td>
<td>48</td>
<td>48</td>
<td>Invalid FT Action frame count.</td>
</tr>
</tbody>
</table>
Table 9 - continued from previous page

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Mapped To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN-VALID_PMKID</td>
<td>49</td>
<td>49</td>
<td>Invalid pairwise master key identifier (PMKID).</td>
</tr>
<tr>
<td>IN-VALID_MDE</td>
<td>50</td>
<td>50</td>
<td>Invalid MDE.</td>
</tr>
<tr>
<td>IN-VALID_FTE</td>
<td>51</td>
<td>51</td>
<td>Invalid FTE</td>
</tr>
<tr>
<td>TRANSMISSION_LINK_ESTABLISHMENT_FAILED</td>
<td>67</td>
<td>67</td>
<td>Transmission link establishment in alternative channel failed.</td>
</tr>
<tr>
<td>ALTERATIVE_CHANNEL_OCCUPIED</td>
<td>68</td>
<td>68</td>
<td>The alternative channel is occupied.</td>
</tr>
<tr>
<td>BEACON_TIMEOUT</td>
<td>200</td>
<td>reserved</td>
<td>Espressif-specific Wi-Fi reason code: when the station loses N beacons continuously, it will disrupt the connection and report this reason.</td>
</tr>
<tr>
<td>NO_AP_FOUND</td>
<td>201</td>
<td>reserved</td>
<td>Espressif-specific Wi-Fi reason code: when the station fails to scan the target AP, this reason code will be reported.</td>
</tr>
<tr>
<td>AUTH FAIL</td>
<td>202</td>
<td>reserved</td>
<td>Espressif-specific Wi-Fi reason code: the authentication fails, but not because of a timeout.</td>
</tr>
<tr>
<td>ASSOC FAIL</td>
<td>203</td>
<td>reserved</td>
<td>Espressif-specific Wi-Fi reason code: the association fails, but not because of ASSOC_EXPIRE or ASSOC_TOOMANY.</td>
</tr>
<tr>
<td>HANDSHAKE_TIMEOUT</td>
<td>204</td>
<td>reserved</td>
<td>Espressif-specific Wi-Fi reason code: the handshake fails for the same reason as that in WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT.</td>
</tr>
<tr>
<td>CONNECTION_FAIL</td>
<td>205</td>
<td>reserved</td>
<td>Espressif-specific Wi-Fi reason code: the connection to the AP has failed.</td>
</tr>
</tbody>
</table>

Wi-Fi Reason code related to wrong password

The table below shows the Wi-Fi reason-code may related to wrong password.

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4WAY_HANDSHAKE_TIMEOUT</td>
<td>15</td>
<td>Four-way handshake times out. Setting wrong password when STA connecting to an encrypted AP.</td>
</tr>
<tr>
<td>NO_AP_FOUND</td>
<td>201</td>
<td>This may related to wrong password in the two scenarios:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Setting password when STA connecting to an unencrypted AP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Does not set password when STA connecting to an encrypted AP.</td>
</tr>
<tr>
<td>HANDSHAKE_TIMEOUT</td>
<td>204</td>
<td>Four-way handshake fails.</td>
</tr>
</tbody>
</table>

Wi-Fi Reason code related to low RSSI

The table below shows the Wi-Fi reason-code may related to low RSSI.
### 4.31.11 ESP32-S2 Wi-Fi Station Connecting When Multiple APs Are Found

This scenario is similar as ESP32-S2 Wi-Fi Station Connecting Scenario. The difference is that the station will not raise the event `WIFI_EVENT_STA_DISCONNECTED` unless it fails to connect all of the found APs.

### 4.31.12 Wi-Fi Reconnect

The station may disconnect due to many reasons, e.g., the connected AP is restarted. It is the application’s responsibility to reconnect. The recommended reconnection strategy is to call `esp_wifi_connect()` on receiving event `WIFI_EVENT_STA_DISCONNECTED`.

Sometimes the application needs more complex reconnection strategy:

- If the disconnect event is raised because the `esp_wifi_disconnect()` is called, the application may not want to do the reconnection.
- If the `esp_wifi_scan_start()` may be called at anytime, a better reconnection strategy is necessary. Refer to Scan When Wi-Fi Is Connecting.

Another thing that need to be considered is that the reconnection may not connect the same AP if there are more than one APs with the same SSID. The reconnection always select current best APs to connect.

### 4.31.13 Wi-Fi Beacon Timeout

The beacon timeout mechanism is used by ESP32-S2 station to detect whether the AP is alive or not. If the station does not receive the beacon of the connected AP within the inactive time, the beacon timeout happens. The application can set inactive time via API `esp_wifi_set_inactive_time()`.

After the beacon times out, the station sends 5 probe requests to the AP. If still no probe response or beacon is received from AP, the station disconnects from the AP and raises the event `WIFI_EVENT_STA_DISCONNECTED`.

It should be considered that the timer used for beacon timeout will be reset during the scanning process. It means that the scan process will affect the triggering of the event `WIFI_EVENT_STA_BEACON_TIMEOUT`.

### 4.31.14 ESP32-S2 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.

---

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_AP_FOUND</td>
<td>201</td>
<td>The station fails to scan the target AP due to low RSSI</td>
</tr>
<tr>
<td>HAND_SHAKE</td>
<td>204</td>
<td>Four-way handshake fails.</td>
</tr>
<tr>
<td>Mode</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>WIFI_MODE_NULL</td>
<td>NULL mode: 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 station and the AP without calling <code>esp_wifi_deinit()</code> to unload the whole Wi-Fi driver.</td>
<td></td>
</tr>
<tr>
<td>WIFI_MODE_STA</td>
<td>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>, the station will connect to the target AP.</td>
<td></td>
</tr>
<tr>
<td>WIFI_MODE_AP</td>
<td>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 broad-casting beacons, and the AP is ready to get connected to other stations.</td>
<td></td>
</tr>
<tr>
<td>WIFI_MODE_AP_STA</td>
<td>Station/AP coexistence mode: in this mode, <code>esp_wifi_start()</code> will simultaneously initialize 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 ESP station is connected to, has higher priority over the ESP AP channel.</td>
<td></td>
</tr>
</tbody>
</table>

**Station Basic Configuration**

API `esp_wifi_set_config()` can be used to configure the station. And the configuration will be stored in NVS. The table below describes the fields in detail.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssid</td>
<td>This is the SSID of the target AP, to which the station wants to connect.</td>
</tr>
<tr>
<td>password</td>
<td>Password of the target AP.</td>
</tr>
<tr>
<td>scan_method</td>
<td>For WIFI_FAST_SCAN scan, the scan ends when the first matched AP is found. For WIFI_ALL_CHANNEL_SCAN, the scan finds all matched APs on all channels. The default scan is WIFI_FAST_SCAN.</td>
</tr>
<tr>
<td>bssid_set</td>
<td>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.</td>
</tr>
<tr>
<td>bssid</td>
<td>This is valid only when bssid_set is 1; see field “bssid_set”.</td>
</tr>
<tr>
<td>channel</td>
<td>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 the channel 1 ~ N but skip the specific channel to find the target AP. For example, if the channel is 3, the scan order will be 3, 1, 2, 4, ..., N. If you do not know which channel the target AP is running on, set it to 0.</td>
</tr>
<tr>
<td>sort_method</td>
<td>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, and the AP with the best signal will be connected firstly. For example, the station wants to connect an AP whose SSID is “apxx”. If the scan finds two APs whose SSID equals to “apxx”, and the first AP’s signal is -90 dBm while the second AP’s signal is -30 dBm, the station connects the second AP firstly, and it would not 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. For example, the station wants to connect an AP whose SSID is “apxx”. If the scan finds two APs whose SSID is “apxx”, and the security of the first found AP is open while the second one is WPA2, the station connects to the second AP firstly, and it would not connect the first one unless it fails to connect the second one.</td>
</tr>
<tr>
<td>threshold</td>
<td>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 discarded. If the RSSI is set to 0, it means the default threshold and the default RSSI threshold are -127 dBm. If the authmode threshold is set to 0, it means the default threshold and the default authmode threshold are open.</td>
</tr>
</tbody>
</table>
Attention: WEP/WPA security modes are deprecated in IEEE 802.11-2016 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. And the configuration will be stored in NVS. The table below describes the fields in detail.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssid</td>
<td>SSID of AP; if the ssid[0] is 0xFF and ssid[1] is 0xFF, the AP defaults the SSID to ESP_aabbcc, where “aabbcc” is the last three bytes of the AP MAC.</td>
</tr>
<tr>
<td>password</td>
<td>Password of AP; if the auth mode is WIFI_AUTH_OPEN, this field will be ignored.</td>
</tr>
<tr>
<td>ssid_len</td>
<td>Length of SSID; if ssid_len is 0, check the SSID until there is a termination character. If ssid_len &gt; 32, change it to 32; otherwise, set the SSID length according to ssid_len.</td>
</tr>
<tr>
<td>channel</td>
<td>Channel of AP; if the channel is out of range, the Wi-Fi driver defaults to channel 1. So, please make sure the channel is within the required range. For more details, refer to Wi-Fi Country Code.</td>
</tr>
<tr>
<td>authmode</td>
<td>Auth mode of ESP AP; currently, ESP AP does not support AUTH_WEP. If the authmode is an invalid value, AP defaults the value to WIFI_AUTH_OPEN.</td>
</tr>
<tr>
<td>ssid_hidden</td>
<td>If ssid_hidden is 1, AP does not broadcast the SSID; otherwise, it does broadcast the SSID.</td>
</tr>
<tr>
<td>max_connection</td>
<td>The max number of stations allowed to connect in, the default value is 10. ESP Wi-Fi supports up to 15 (ESP_WIFI_MAX_CONN_NUM) Wi-Fi connections. Please note that ESP AP and ESP-NOW share the same encryption hardware keys, so the max_connection parameter will be affected by the CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM. The total number of encryption hardware keys is 17, if CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM &lt;= 2, the max_connection can be set up to 15, otherwise the max_connection can be set up to (17 - CONFIG_ESP_WIFI_ESPNOW_MAX_ENCRYPT_NUM).</td>
</tr>
<tr>
<td>beacon_interval</td>
<td>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.</td>
</tr>
</tbody>
</table>

Wi-Fi Protocol Mode

Currently, the ESP-IDF supports the following protocol modes:
### Protocol Mode Description

<table>
<thead>
<tr>
<th>Protocol Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.11b</td>
<td>Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B)</code> to set the station/AP to 802.11b-only mode.</td>
</tr>
<tr>
<td>802.11bg</td>
<td>Call `esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B</td>
</tr>
<tr>
<td>802.11g</td>
<td>Call `esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B</td>
</tr>
<tr>
<td>802.11bgn</td>
<td>Call `esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B</td>
</tr>
<tr>
<td>802.11gn</td>
<td>Call `esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B</td>
</tr>
<tr>
<td>802.11 BGNLR</td>
<td>Call `esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B</td>
</tr>
<tr>
<td>802.11 LR</td>
<td>Call <code>esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_LR)</code> to set the station/AP only to the LR mode. <strong>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 ESP device.</strong></td>
</tr>
</tbody>
</table>

### Long Range (LR)

Long Range (LR) mode is an Espressif-patented Wi-Fi mode which can achieve a one-kilometer line of sight range. Compared to the traditional 802.11b mode, it has better reception sensitivity, stronger anti-interference ability, and longer transmission distance.

**LR Compatibility** Since LR is Espressif-unique Wi-Fi mode, only ESP32 chip series devices (except ESP32-C2) can transmit and receive the LR data. In other words, the ESP32 chip series devices (except ESP32-C2) should NOT transmit the data in LR data rate if the connected device does not support LR. The application can achieve this by configuring a suitable Wi-Fi mode. If the negotiated mode supports LR, the ESP32 chip series devices (except ESP32-C2) may transmit data in LR rate. Otherwise, ESP32 chip series devices (except ESP32-C2) will transmit all data in the traditional Wi-Fi data rate.

The following table depicts the Wi-Fi mode negotiation:

<table>
<thead>
<tr>
<th>APSTA BGN</th>
<th>BGN</th>
<th>BG</th>
<th>B</th>
<th>BGNLR</th>
<th>BGLR</th>
<th>BLR</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGN</td>
<td>BGN</td>
<td>BG</td>
<td>B</td>
<td>BGN</td>
<td>B</td>
<td>B</td>
<td>*</td>
</tr>
<tr>
<td>BG</td>
<td>BG</td>
<td>BG</td>
<td>B</td>
<td>BG</td>
<td>BG</td>
<td>B</td>
<td>*</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>*</td>
</tr>
<tr>
<td>BGNLR</td>
<td></td>
<td></td>
<td></td>
<td>BGNLR</td>
<td>BGLR</td>
<td>BLR</td>
<td>LR</td>
</tr>
<tr>
<td>BGLR</td>
<td></td>
<td></td>
<td></td>
<td>BGLR</td>
<td>BGLR</td>
<td>BLR</td>
<td>LR</td>
</tr>
<tr>
<td>BLR</td>
<td></td>
<td></td>
<td></td>
<td>BLR</td>
<td>BLR</td>
<td>BLR</td>
<td>LR</td>
</tr>
<tr>
<td>LR</td>
<td></td>
<td></td>
<td></td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
</tr>
</tbody>
</table>
In the 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, the following conclusions can be drawn:

- For LR-enabled AP of ESP32-S2, it is incompatible with traditional 802.11 mode, because the beacon is sent in LR mode.
- For LR-enabled station of ESP32-S2 whose mode is NOT LR-only mode, it is compatible with traditional 802.11 mode.
- If both station and AP are ESP32 series chips devices (except ESP32-C2) and both of them have enabled LR mode, the negotiated mode supports LR.

If the negotiated Wi-Fi mode supports both traditional 802.11 mode and LR mode, it is the Wi-Fi driver’s responsibility to automatically select the best data rate in different Wi-Fi modes and the application can ignore 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 transmission impact in LR mode is similar to that in 802.11b mode.

**LR Transmission Distance**  The reception sensitivity gain of LR is about 4 dB larger than that of the traditional 802.11b 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 Mbps and 1/4 Mbps.

**When to Use LR**  The general conditions for using LR are:

- Both the AP and station are Espressif devices.
- Long distance Wi-Fi connection and data transmission is required.
- Data throughput requirements are very small, such as remote device control.

**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 2.4 GHz RF operating regulations before configuring these fields.
### Field | Description
--- | ---
cc[3] | Country code string. This attribute identifies the country or noncountry entity in which the station/AP is operating. If it is a country, the first two octets of this string is the two-character country info as described in the document ISO/IEC3166-1. The third octet is one of the following:
- an ASCII space character, which means the regulations under which the station/AP is operating encompass all environments for the current frequency band in the country.
- an ASCII ‘O’ character, which means the regulations under which the station/AP is operating are for an outdoor environment only.
- an ASCII ‘I’ character, which means the regulations under which the station/AP is operating are for an indoor environment only.
- an ASCII ‘X’ character, which means the station/AP is operating under a non-country entity. The first two octets of the noncountry entity is two ASCII ‘XX’ characters.
- the binary representation of the Operating Class table number currently in use. Refer to Annex E of IEEE Std 802.11-2020.

schan | Start channel. It is the minimum channel number of the regulations under which the station/AP can operate.
nchan | Total number of channels as per the regulations. For example, if the schan=1, nchan=13, then the station/AP can send data from channel 1 to 13.
policy | Country policy. This field controls which country info will be used if the configured country info is in conflict with the connected AP’s. For more details on related policies, see the following section.

The default country info is:

```c
country_t config = {
  .cc = "01",
  .schan = 1,
  .nchan = 11,
  .policy = WIFI_COUNTRY_POLICY_AUTO,
};
```

If the Wi-Fi 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:

```c
country_t config = {
  .cc = "JP",
  .schan = 1,
  .nchan = 14,
  .policy = WIFI_COUNTRY_POLICY_AUTO,
};
```

but the connected AP has country info:

```c
country_t config = {
  .cc = "CN",
  .schan = 1,
  .nchan = 13,
};
```

then country info of connected AP’s is used.

The following table depicts which country info is used in different Wi-Fi modes and different country policies, and it also describes the impact on active scan.
### Wi-Fi Mode

<table>
<thead>
<tr>
<th>Wi-Fi Mode</th>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Station</td>
<td>WIFI_COUNTRY_POLICY_AUTO</td>
<td>If the connected AP has country IE in its beacon, the country info equals to the country info in beacon. Otherwise, use the default country info. For scan: Use active scan from 1 to 11 and use passive scan from 12 to 14. Always keep in mind that if an AP with hidden SSID and station 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.</td>
</tr>
<tr>
<td>Station</td>
<td>WIFI_COUNTRY_POLICY_MANUAL</td>
<td>Always use the configured country info. For scan: Use active scan from schan to schan+nchan-1.</td>
</tr>
<tr>
<td>AP</td>
<td>WIFI_COUNTRY_POLICY_AUTO</td>
<td>Same as station mode with policy WIFI_COUNTRY_POLICY_AUTO. AP: If the station does not connect to any external AP, the AP uses the configured country info. If the station connects to an external AP, the AP has the same country info as the station.</td>
</tr>
<tr>
<td>Station/AP-coexistence</td>
<td>WIFI_COUNTRY_POLICY_AUTO</td>
<td>Same as station mode with policy WIFI_COUNTRY_POLICY_AUTO. AP: Same as AP mode with policy WIFI_COUNTRY_POLICY_MANUAL.</td>
</tr>
<tr>
<td>Station/AP-coexistence</td>
<td>WIFI_COUNTRY_POLICY_MANUAL</td>
<td>Same as station mode with policy WIFI_COUNTRY_POLICY_MANUAL. AP: Same as AP mode with policy WIFI_COUNTRY_POLICY_MANUAL.</td>
</tr>
</tbody>
</table>

### Home Channel

In AP mode, the home channel is defined as the AP channel. In station mode, home channel is defined as the channel of AP which the station is connected to. In station/AP-coexistence mode, the home channel of AP and station must be the same, and if they are different, the station’s home channel is always in priority. For example, assume that the AP is on channel 6, and the station connects to an AP whose channel is 9. Since the station’s home channel has higher priority, the AP needs to switch its channel from 6 to 9 to make sure that it has the same home channel as the station. While switching channel, the ESP32-S2 in AP mode will notify the connected stations about the channel migration using a Channel Switch Announcement (CSA). Station that supports channel switching will transit without disconnecting and reconnecting to the AP.

### Wi-Fi Vendor IE Configuration

By default, all Wi-Fi management frames are processed by the Wi-Fi driver, and the application can ignore them. However, some applications 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-S2, `esp_wifi_set_vendor_ie()` and `esp_wifi_set_vendor_ie_cb()` are responsible for this kind of tasks.

### 4.31.15 Wi-Fi Easy Connect™ (DPP)

Wi-Fi Easy Connect™ (or Device Provisioning Protocol) is a secure and standardized provisioning protocol for configuring Wi-Fi devices. More information can be found in `esp_dpp`. 
WPA2-Enterprise

WPA2-Enterprise is the secure authentication mechanism for enterprise wireless networks. It uses RADIUS server for authentication of network users before connecting to the Access Point. The authentication process is based on 802.1X policy and comes with different Extended Authentication Protocol (EAP) methods such as TLS, TTLS, and PEAP. RADIUS server authenticates the users based on their credentials (username and password), digital certificates, or both. When ESP32-S2 in station mode tries to connect an AP in enterprise mode, it sends authentication request to AP which is sent to RADIUS server by AP for authenticating the station. Based on different EAP methods, the parameters can be set in configuration which can be opened using idf.py menuconfig. WPA2_Enterprise is supported by ESP32-S2 only in station mode.

For establishing a secure connection, AP and station negotiate and agree on the best possible cipher suite to be used. ESP32-S2 supports 802.1X/EAP (WPA) method of AKM and Advanced encryption standard with Counter Mode Cipher Block Chaining Message Authentication protocol (AES-CCM) cipher suite. It also supports the cipher suites supported by mbedtls if USE_MBEDTLS_CRYPTO flag is set.

ESP32-S2 currently supports the following EAP methods:

- **EAP-TLS**: This is a certificate-based method and only requires SSID and EAP-IDF.
- **PEAP**: This is a Protected EAP method. Username and Password are mandatory.
- **EAP-TTLS**: This is a credential-based method. Only server authentication is mandatory while user authentication is optional. Username and Password are mandatory. It supports different Phase2 methods, such as:
  - **PAP**: Password Authentication Protocol.
  - **CHAP**: Challenge Handshake Authentication Protocol.
  - **MSCHAP** and **MSCHAP-V2**.
- **EAP-FAST**: This is an authentication method based on Protected Access Credentials (PAC) which also uses identity and password. Currently, USE_MBEDTLS_CRYPTO flag should be disabled to use this feature.

Detailed information on creating certificates and how to run wpa2_enterprise example on ESP32-S2 can be found in wifi/wifi_enterprise.

4.31.16 Wi-Fi Aware™ (NAN)

Wi-Fi Aware™ or NAN (Neighbor Awareness Networking) is a protocol that allows Wi-Fi devices to discover services in their proximity. NAN uses direct device-to-device communication and does not require any Internet or AP connection.

Multiple NAN devices in the vicinity will form a NAN cluster which allows them to communicate with each other. NAN devices in a cluster synchronise their clocks and listen to each other periodically on Channel 6. Devices can advertise (Publish) or seek for (Subscribe) services within their NAN Cluster using Service Discovery protocols. Matching of services is done by service name and optionally matching filters. Once a Subscriber gets a match with a Publisher, it can either send a message (Follow-up) or establish a datapath (NDP) with the Publisher. After NDP is setup both devices will obtain an IPv6 address and can use it for communication.

Please note that NAN Datapath security is not supported i.e., the data packets will go out unencrypted. NAN uses a separate interface for Discovery and Datapath, which is other than that used for STA and AP. NAN operates in standalone mode, which means co-existence with STA or AP interface is not supported.

Refer to ESP-IDF examples examples/wifi/wifi_aware/nan_publisher/README.md and examples/wifi/wifi_aware/nan_subscriber/README.md to setup a NAN Publisher and Subscriber.

4.31.17 Wireless Network Management

Wireless Network Management allows client devices to exchange information about the network topology, including information related to RF environment. This makes each client network-aware, facilitating overall improvement in the performance of the wireless network. It is part of 802.11v specification. It also enables the client to support Network assisted Roaming. - Network assisted Roaming: Enables WLAN to send messages to associated clients, resulting clients to associate with APs with better link metrics. This is useful for both load balancing and in directing poorly connected clients.
Current implementation of 802.11v includes support for BSS transition management frames.

### 4.31.18 Radio Resource Measurement

Radio Resource Measurement (802.11k) is intended to improve the way traffic is distributed within a network. In a WLAN, each device normally connects to the access point (AP) that provides the strongest signal. Depending on the number and geographic locations of the subscribers, this arrangement can sometimes lead to excessive demand on one AP and underutilization of others, resulting in degradation of overall network performance. In a network conforming to 802.11k, if the AP having the strongest signal is loaded to its full capacity, a wireless device can be moved to one of the underutilized APs. Even though the signal may be weaker, the overall throughput is greater because more efficient use is made of the network resources.

Current implementation of 802.11k includes support for beacon measurement report, link measurement report, and neighbor request. Refer ESP-IDF example examples/wifi/roaming/README.md to set up and use these APIs. Example code only demonstrates how these APIs can be used, and the application should define its own algorithm and cases as required.

### 4.31.19 Fast BSS Transition

Fast BSS transition (802.11R FT), is a standard to permit continuous connectivity aboard wireless devices in motion, with fast and secure client transitions from one Basic Service Set (abbreviated BSS, and also known as a base station or more colloquially, an access point) to another performed in a nearly seamless manner avoiding 802.11 4 way handshake. 802.11R specifies transitions between access points by redefining the security key negotiation protocol, allowing both the negotiation and requests for wireless resources to occur in parallel. The key derived from the server to be cached in the wireless network, so that a reasonable number of future connections can be based on the cached key, avoiding the 802.1X process.

ESP32-S2 station supports FT for WPA2-PSK networks. Do note that ESP32-S2 station only support FT over the air protocol only.

A config option `CONFIG_ESP_WIFI_11R_SUPPORT` and configuration parameter `ft_enabled` in `wifi_sta_config_t` is provided to enable 802.11R support for station. Refer ESP-IDF example examples/wifi/roaming/README.md for further details.

### 4.31.20 Wi-Fi Location

Wi-Fi Location will improve the accuracy of a device’s location data beyond the Access Point, which will enable creation of new and feature-rich applications and services such as geo-fencing, network management, and navigation. One of the protocols used to determine the device location with respect to the Access Point is Fine Timing Measurement which calculates Time-of-Flight of a Wi-Fi frame.

**Fine Timing Measurement (FTM)**

FTM is used to measure Wi-Fi Round Trip Time (Wi-Fi RTT) which is the time a Wi-Fi signal takes to travel from a device to another device and back again. Using Wi-Fi RTT, the distance between the devices can be calculated with a simple formula of \( RTT \times c / 2 \), where \( c \) is the speed of light.

FTM uses timestamps given by Wi-Fi interface hardware at the time of arrival or departure of frames exchanged between a pair of devices. One entity called FTM Initiator (mostly a station device) discovers the FTM Responder (can be a station or an Access Point) and negotiates to start an FTM procedure. The procedure uses multiple Action frames sent in bursts and its ACK’s to gather the timestamps data. FTM Initiator gathers the data in the end to calculate an average Round-Trip-Time.

ESP32-S2 supports FTM in below configuration:

- ESP32-S2 as FTM Initiator in station mode.
- ESP32-S2 as FTM Responder in AP mode.
Distance measurement using RTT is not accurate, and factors such as RF interference, multi-path travel, antenna orientation, and lack of calibration increase these inaccuracies. For better results, it is suggested to perform FTM between two ESP32 chip series devices (except ESP32-C2) as station and AP.

Refer to ESP-IDF example examples/wifi/ftm/README.md for steps on how to set up and perform FTM.

4.31.21 ESP32-S2 Wi-Fi Power-saving Mode

Station Sleep

Currently, ESP32-S2 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-saving modes. In minimum power-saving mode, station wakes up every DTIM to receive beacon. Broadcast data will not be lost because it is transmitted after DTIM. However, it cannot save much more power if DTIM is short for DTIM is determined by AP.

In maximum power-saving mode, station wakes up in every listen interval to receive beacon. This listen interval can be set to be 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-saving mode or esp_wifi_set_ps(WIFI_PS_MAX_MODEM) to enable Modem-sleep maximum power-saving 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 mode entirely. Disabling it increases power consumption, but minimizes the delay in receiving Wi-Fi data in real time. When Modem-sleep mode is enabled, the delay in receiving Wi-Fi data may be the same as the DTIM cycle (minimum power-saving mode) or the listening interval (maximum power-saving mode).

The default Modem-sleep mode is WIFI_PS_MIN_MODEM.

AP Sleep

Currently, ESP32-S2 AP does not support all of the power-saving feature defined in Wi-Fi specification. To be specific, the AP only caches unicast data for the stations connect to this AP, but does not cache the multicast data for the stations. If stations connected to the ESP32-S2 AP are power-saving enabled, they may experience multicast packet loss.

In the future, all power-saving features will be supported on ESP32-S2 AP.

Disconnected State Sleep

Disconnected state is the duration without Wi-Fi connection between esp_wifi_start() to esp_wifi_stop().

Currently, ESP32-S2 Wi-Fi supports sleep mode in disconnected state if running at station mode. This feature could be configured by Menuconfig choice CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE.

If CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE is enabled, RF, PHY and BB would be turned off in disconnected state when IDLE. The current would be same with current at modem-sleep.

The choice CONFIG_ESP_WIFI_STA_DISCONNECTED_PM_ENABLE would be selected by default, while it would be selected forcefully in Menuconfig at coexistence mode.
Chapter 4. API Guides

**Connectionless Modules Power-saving**

Connectionless modules are those Wi-Fi modules not relying on Wi-Fi connection, e.g ESP-NOW, DPP, FTM. These modules start from `esp_wifi_start()`, working until `esp_wifi_stop()`.

Currently, if ESP-NOW works at station mode, it's supported to sleep at both connected state and disconnected state.

**Connectionless Modules TX**  For each connectionless module, its supported to TX at any sleeping time without any extra configuration.

Meanwhile, `esp_wifi_80211_tx()` is supported at sleep as well.

**Connectionless Modules RX**  For each connectionless module, two parameters shall be configured to RX at sleep, which are Window and Interval.

At the start of Interval time, RF, PHY, BB would be turned on and kept for Window time. Connectionless Module could RX in the duration.

**Interval**

- There is only one Interval. Its configured by `esp_wifi_set_connectionless_interval()`. The unit is milliseconds.
- The default value of Interval is `ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE`.
- Event `WIFI_EVENT_CONNECTIONLESS MODULE_WAKE_INTERVAL_START` would be posted at the start of Interval. Since Window also starts at that moment, it's recommended to TX in that event.
- At connected state, the start of Interval would be aligned with TBTT.

**Window**

- Each connectionless module has its own Window after start. Connectionless Modules Power-saving would work with the max one among them.
- Window is configured by `module_name_set_wake_window()`. The unit is milliseconds.
- The default value of Window is the maximum.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT_MODE</td>
<td>0</td>
</tr>
<tr>
<td>not used</td>
<td>1 - maximum</td>
</tr>
<tr>
<td>default mode</td>
<td>used periodically (Window &lt; Interval) / used all time (Window ≥ Interval)</td>
</tr>
</tbody>
</table>

**Default Mode**  If Interval is `ESP_WIFI_CONNECTIONLESS INTERVAL_DEFAULT MODE` with non-zero Window, Connectionless Modules Power-saving would work in default mode.

In default mode, RF, PHY, BB would be kept on if no coexistence with non-Wi-Fi protocol.

With coexistence, RF, PHY, BB resources are allocated by coexistence module to Wi-Fi connectionless module and non-Wi-Fi module, using time-division method. In default mode, Wi-Fi connectionless module is allowed to use RF, BB, PHY periodically under a stable performance.

It's recommended to configure Connectionless Modules Power-saving to default mode if there is Wi-Fi connectionless module coexists with non-Wi-Fi module.

**4.31.22 ESP32-S2 Wi-Fi Throughput**

The table below shows the best throughput results gained in Espressif’s lab and in a shielded box.
<table>
<thead>
<tr>
<th>Type/Throughput</th>
<th>Air In Lab</th>
<th>Shield-box</th>
<th>Test Tool</th>
<th>IDF Version (commit ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw 802.11</td>
<td>N/A</td>
<td>130 MBit/s</td>
<td>Internal tool</td>
<td>NA</td>
</tr>
<tr>
<td>Packet RX</td>
<td>N/A</td>
<td>130 MBit/s</td>
<td>Internal tool</td>
<td>NA</td>
</tr>
<tr>
<td>Raw 802.11</td>
<td>N/A</td>
<td>130 MBit/s</td>
<td>Internal tool</td>
<td>NA</td>
</tr>
<tr>
<td>Packet TX</td>
<td>N/A</td>
<td>130 MBit/s</td>
<td>Internal tool</td>
<td>NA</td>
</tr>
<tr>
<td>UDP RX</td>
<td>30 MBit/s</td>
<td>70 MBit/s</td>
<td>iperf example</td>
<td>15575346</td>
</tr>
<tr>
<td>UDP TX</td>
<td>30 MBit/s</td>
<td>50 MBit/s</td>
<td>iperf example</td>
<td>15575346</td>
</tr>
<tr>
<td>TCP RX</td>
<td>20 MBit/s</td>
<td>32 MBit/s</td>
<td>iperf example</td>
<td>15575346</td>
</tr>
<tr>
<td>TCP TX</td>
<td>20 MBit/s</td>
<td>37 MBit/s</td>
<td>iperf example</td>
<td>15575346</td>
</tr>
</tbody>
</table>

When the throughput is tested by iperf example, the sdkconfig is examples/wifi/iperf/sdkconfig.defaults.esp32s2.

### 4.31.2.3 Wi-Fi 80211 Packet Send

The `esp_wifi_80211_tx()` API can be used to:

- Send the beacon, probe request, probe response, and 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 Wi-Fi hardware must be initialized before `esp_wifi_80211_tx()` is called. In ESP32-S2, 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

- The default data rate is 1 Mbps.
- Can set any rate through `esp_wifi_config_80211_tx_rate()` API.
- Can set any bandwidth through `esp_wifi_set_bandwidth()` API.

#### Side-Effects to Avoid in Different Scenarios

Theoretically, if the side-effects the API imposes on the Wi-Fi driver or other stations/APs are not considered, a raw 802.11 packet can be sent over the air with any destination MAC, any source MAC, any BSSID, or any other types of packet. However, robust or useful applications should avoid such side-effects. The table below provides some tips and recommendations on how to avoid the side-effects of `esp_wifi_80211_tx()` in different scenarios.
### Scenario Description

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
</table>
| No Wi-Fi connection       | In this scenario, no Wi-Fi connection is set up, so there are no side-effects on the Wi-Fi driver. If `en_sys_seq`=True, the Wi-Fi driver is responsible for the sequence control. If `en_sys_seq=False`, the application needs to ensure that the buffer has the correct sequence. Theoretically, the MAC address can be any address. However, this may impact other stations/APs with the same MAC/BSSID. Side-effect example#1 The application calls `esp_wifi_80211_tx()` to send a beacon with BSSID == mac_x in AP mode, but the mac_x is not the MAC of the AP interface. Moreover, there is another AP, e.g., “other-AP”, whose BSSID is mac_x. 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 `esp_wifi_80211_tx()`. To avoid the above-mentioned side-effects, it is recommended that:  
  - If `esp_wifi_80211_tx()` 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 `esp_wifi_80211_tx()` 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.  
  The recommendations above are only for avoiding side-effects and can be ignored when there are good reasons. |
| Have Wi-Fi connection     | 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 `en_sys_seq` need to be true, otherwise ESP_ERR_INVALID_ARG is returned.  
  The MAC-address recommendations in the “No Wi-Fi connection” scenario also apply to this scenario.  
  If the Wi-Fi mode is station mode, the MAC address1 is the MAC of AP to which the station is connected, and the MAC address2 is the MAC of station interface, it is said that the packet is sent from the station to AP. Otherwise, if the Wi-Fi is in AP mode, the MAC address1 is the MAC of the station that connects to this AP, and the MAC address2 is the MAC of AP interface, it is said that the packet is sent from the AP to station. To avoid conflicting with Wi-Fi connections, the following checks are applied:  
  - If the packet type is data and is sent from the station to AP, the ToDS bit in IEEE 80211 frame control should be 1 and the FromDS bit should be 0. Otherwise, the packet will be discarded by Wi-Fi driver.  
  - If the packet type is data and is sent from the AP to station, the ToDS bit in IEEE 80211 frame control should be 0 and the FromDS bit should be 1. Otherwise, the packet will be discarded by Wi-Fi driver.  
  - If the packet is sent from station to AP or from AP to station, the Power Management, More Data, and Re-Transmission bits should be 0. Otherwise, the packet will be discarded by Wi-Fi driver.  
  ESP_ERR_INVALID_ARG is returned if any check fails. |

#### 4.31.24 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, and AMSDU.
- 802.11 MIMO frame, for MIMO frame, the sniffer only dumps the length of the frame.
- 802.11 Control frame.
- 802.11 CRC error frame.

The following packets will NOT be dumped to the application:

- Other 802.11 error frames.
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`, `WIFI_MODE_STA`, `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, the sniffer should be enabled only if the station/AP Wi-Fi connection does not experience heavy traffic.

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.

### 4.31.25 Wi-Fi Multiple Antennas

The Wi-Fi multiple antennas selecting can be depicted as following picture:

```
<table>
<thead>
<tr>
<th>Enabled</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Antenna 0</td>
<td></td>
</tr>
</tbody>
</table>
```

ESP32-S2 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. For example, the value `0b1011` means the antenna 11 is selected. The default value of `antenna_select[3:0]` is `0b0000`, which means the antenna 0 is selected by default.

Up to four GPIOs are connected to the four active high antenna select pins. ESP32-S2 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.

For the specific implementation of the antenna switch, there may be illegal values in `antenna_select[0:3]`. It means that ESP32-S2 may support less than sixteen antennas through the switch. For example, ESP32-WROOM-DA which uses RTC6603SP as the antenna switch, supports two antennas. Two GPIOs are connected to two active high antenna selection inputs. The value `0b01` means the antenna 0 is selected, the value `0b10` means the antenna 1 is selected. Values `0b00` and `0b11` are illegal.

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, Wi-Fi automatically selects the enabled antenna that has better signal.

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. For example, the RX antenna starts to switch if
the RSSI is lower than -65 dBm or another antenna has better signal. 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.

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.
- When the TX antenna mode or RX antenna mode is configured to `WIFI_ANT_MODE_AUTO` the switching mode will easily trigger the switching phase, as long as there is deterioration of the RF signal. So in situations where the RF signal is not stable, the antenna switching will occur frequently, resulting in an RF performance that may not meet expectations.
- Currently, Bluetooth® does not support the multiple antennas feature, so please do not use multiple antennas related APIs.

Following is the recommended scenarios to use the multiple antennas:

- The applications can always choose to select a specified antenna or implement their own antenna selecting algorithm, e.g., selecting the antenna mode based on the information collected by the application. Refer to ESP-IDF example `examples/wifi/antenna/README.md` for the antenna selecting algorithm design.
- 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:

```c
wifi_ant_gpio_config_t ant_gpio_config = {
    .gpio_cfg[0] = { .gpio_select = 1, .gpio_num = 20 },
    .gpio_cfg[1] = { .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:

```c
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
};
```

### 4.31.26 Wi-Fi Channel State Information

Channel state information (CSI) refers to the channel information of a Wi-Fi connection. In ESP32-S2, 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 are shown in the following table.
### Chapter 4. API Guides

#### Channel

<table>
<thead>
<tr>
<th>Channel</th>
<th>Secondary Channel</th>
<th>None</th>
<th>Below</th>
<th>Above</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packets Information</td>
<td>Signal Mode</td>
<td>HT</td>
<td>20 MHz</td>
<td>40 MHz</td>
</tr>
<tr>
<td>Channel Bandwidth</td>
<td>STBC</td>
<td>Non</td>
<td>STBC</td>
<td>STBC</td>
</tr>
<tr>
<td>Sub-carrier Index</td>
<td>LLTF</td>
<td>0<del>31, -32</del>1</td>
<td>0<del>31, -32</del>1</td>
<td>0~63</td>
</tr>
<tr>
<td></td>
<td>HT-LTF</td>
<td>*</td>
<td>0<del>31, -32</del>1</td>
<td>0~62</td>
</tr>
<tr>
<td></td>
<td>STBC-HT-LTF</td>
<td>*</td>
<td>0<del>31, -32</del>1</td>
<td>0~62</td>
</tr>
<tr>
<td>Total bytes</td>
<td>128</td>
<td>256</td>
<td>384</td>
<td>128</td>
</tr>
</tbody>
</table>

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-S2.
- More information like RSSI, noise floor of RF, receiving time and antenna is in the `rx_ctrl` field.

When imaginary part and real part data of sub-carrier are used, please refer to the table below.

<table>
<thead>
<tr>
<th>PHY standard</th>
<th>Sub-carrier range</th>
<th>Pilot sub-carrier</th>
<th>Sub-carrier (total/data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.11a/g</td>
<td>-26 to +26</td>
<td>-21, -7, +7, +21</td>
<td>52 total, 48 usable</td>
</tr>
<tr>
<td>802.11n, 20 MHz</td>
<td>-28 to +28</td>
<td>-21, -7, +7, +21</td>
<td>56 total, 52 usable</td>
</tr>
<tr>
<td>802.11n, 40 MHz</td>
<td>-57 to +57</td>
<td>-53, -25, -11, +11, +25, +53</td>
<td>114 total, 108 usable</td>
</tr>
</tbody>
</table>

**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 -200 ns, 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 pi.
- 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 \((61 + 60) \times 2 + 2 = 244\). The result is aligned to four bytes, and the last two bytes are invalid.

4.31.27 Wi-Fi Channel State Information Configure

To use Wi-Fi CSI, the following steps need to be done.

- Select Wi-Fi CSI in menuconfig. Go to `Menuconfig > Components config > Wi-Fi > Wi-Fi 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()`.

4.31.28 Wi-Fi HT20/40

ESP32-S2 supports Wi-Fi bandwidth HT20 or HT40 and does not 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-S2 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 is 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 is HT40 if the AP and one of the stations support HT40, otherwise it is HT20.

In station/AP coexist mode, the station/AP can configure HT20/40 seperately. 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-S2. For example, 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 150 Mbps while it is 72 Mbps 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-S2 device, the performance of HT40 may be degraded. So if the applications need to support same or similar scenarios, it is recommended that the bandwidth is always configured to HT20.

4.31.29 Wi-Fi QoS

ESP32-S2 supports all the mandatory features required in WFA Wi-Fi QoS Certification.

Four ACs (Access Category) are defined in Wi-Fi specification, and each AC has its own priority to access the Wi-Fi channel. Moreover, a map rule is defined to map the QoS priority of other protocol, e.g., 802.11D or TCP/IP precedence is mapped to Wi-Fi AC.

The table below describes how the IP Precedences are mapped to Wi-Fi ACs in ESP32-S2. It also indicates whether the AMPDU is supported for this AC. The table is sorted from high to low priority. That is to say, the AC_VO has the highest priority.
### Chapter 4. API Guides

<table>
<thead>
<tr>
<th>IP Precedence</th>
<th>Wi-Fi AC</th>
<th>Support AMPDU?</th>
</tr>
</thead>
<tbody>
<tr>
<td>6, 7</td>
<td>AC_VO (Voice)</td>
<td>No</td>
</tr>
<tr>
<td>4, 5</td>
<td>AC_VI (Video)</td>
<td>Yes</td>
</tr>
<tr>
<td>3, 0</td>
<td>AC_BE (Best Effort)</td>
<td>Yes</td>
</tr>
<tr>
<td>1, 2</td>
<td>AC_BK (Background)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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:

```c
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 lower priority AC. However, it is not always true. Here are some suggestions about how to use the Wi-Fi QoS:

- Some really important application traffic can be put into the AC_VO queue. But avoid using the AC_VO queue for heavy traffic, as it may impact the management frames which also use this queue. Eventually, it is worth noting that the AC_VO queue does not support AMPDU, and its performance with heavy traffic is no better than other queues.
- Avoid using more than two precedences supported by different AMPDUs, e.g., when socket A uses precedence 0, socket B uses precedence 1, and socket C uses precedence 2. This can be a bad design because it may need much more memory. To be specific, the Wi-Fi driver may generate a Block Ack session for each precedence and it needs more memory if the Block Ack session is set up.

#### 4.31.30 Wi-Fi AMSDU

ESP32-S2 supports receiving and transmitting AMSDU. AMSDU TX is disabled by default, since enable AMSDU TX need more memory. Select `CONFIG_ESP_WIFI_AMSDU_TX_ENABLED` to enable AMSDU Tx feature, it depends on `CONFIG_SPIRAM`.

#### 4.31.31 Wi-Fi Fragment

supports Wi-Fi receiving fragment, but does not support Wi-Fi transmitting fragment.

#### 4.31.32 WPS Enrollee

ESP32-S2 supports WPS enrollee feature in Wi-Fi mode `WIFI_MODE_STA` or `WIFI_MODE_APSTA`. Currently, ESP32-S2 supports WPS enrollee type PBC and PIN.

#### 4.31.33 Wi-Fi Buffer Usage

This section is only about the dynamic buffer configuration.

**Why Buffer Configuration Is Important**

In order to get a high-performance system, consider the memory usage/configuration carefully for the following reasons:

- the available memory in ESP32-S2 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-S2 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 and Wi-Fi RX/TX dynamic buffer number.
• the peak heap memory that the ESP32-S2 LwIP/Wi-Fi may consume depends on a number of factors, such as the maximum TCP/UDP connections that the application may have.
• 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, it is recommended 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:

- $b_{rx}$ the number of dynamic RX buffers that are configured
- $b_{tx}$ the number of dynamic TX buffers that are configured
- $m_{rx}$ the maximum packet size that the Wi-Fi driver can receive
- $m_{tx}$ the maximum packet size that the Wi-Fi driver can send

So, the peak memory that the Wi-Fi driver consumes ($p$) can be calculated with the following formula:

$$p = (b_{rx} \times m_{rx}) + (b_{tx} \times m_{tx})$$

Generally, the dynamic TX long buffers and dynamic TX long long buffers can be ignored, because they are management frames which only have a small impact on the system.

### 4.31.34 How to Improve Wi-Fi Performance

The performance of ESP32-S2 Wi-Fi is affected by many parameters, and there are mutual constraints between each parameter. A proper configuration cannot only improve performance, but also increase available memory for applications and improve stability.

This section briefly explains the operating mode of the Wi-Fi/LwIP protocol stack and the role of each parameter. It also gives several recommended configuration ranks to help choose the appropriate rank according to the usage scenario.

### Protocol Stack Operation Mode

The ESP32-S2 protocol stack is divided into four layers: Application, LwIP, Wi-Fi, and Hardware.

- During receiving, hardware puts the received packet into DMA buffer, and then transfers it into the RX buffer of Wi-Fi and LwIP in turn for related protocol processing, and finally to the application layer. The Wi-Fi RX buffer and the LwIP RX buffer shares the same buffer by default. In other words, the Wi-Fi forwards the packet to LwIP by reference by default.
During sending, the application copies the messages to be sent into the TX buffer of the LwIP layer for TCP/IP encapsulation. The messages will then be passed to the TX buffer of the Wi-Fi layer for MAC encapsulation and wait to be sent.

Parameters

Increasing the size or number of the buffers mentioned above properly can improve Wi-Fi performance. Meanwhile, it will reduce available memory to the application. The following is an introduction to the parameters that users need to configure:

**RX direction:**

- **CONFIG_ESP_WIFI_STATIC_RX_BUFFER_NUM** This parameter indicates the number of DMA buffer at the hardware layer. Increasing this parameter will increase the sender’s one-time receiving throughput, thereby improving the Wi-Fi protocol stack ability to handle burst traffic.

- **CONFIG_ESP_WIFI_DYNAMIC_RX_BUFFER_NUM** This parameter indicates the number of RX buffer in the Wi-Fi layer. Increasing this parameter will improve the performance of packet reception. This parameter needs to match the RX buffers size of the LwIP layer.

- **CONFIG_ESP_WIFI_RX_BA_WIN** This parameter indicates the size of the AMPDU BA Window at the receiving end. This parameter should be configured to the smaller value between twice of **CONFIG_ESP_WIFI_STATIC_RX_BUFFER_NUM** and **CONFIG_ESP_WIFI_DYNAMIC_RX_BUFFER_NUM**.

- **CONFIG_LWIP_TCP_WND_DEFAULT** This parameter represents the RX buffer size of the LwIP layer for each TCP stream. Its value should be configured to the value of **WIFI_DYNAMIC_RX_BUFFER_NUM (KB)** to reach a high and stable performance. Meanwhile, in case of multiple streams, this value needs to be reduced proportionally.

**TX direction:**

- **CONFIG_ESP_WIFI_TX_BUFFER** This parameter indicates the type of TX buffer, it is recommended to configure it as a dynamic buffer, which can make full use of memory.

- **CONFIG_ESP_WIFI_DYNAMIC_TX_BUFFER_NUM** This parameter indicates the number of TX buffer on the Wi-Fi layer. Increasing this parameter will improve the performance of packet sending. The parameter value needs to match the TX buffer size of the LwIP layer.

- **CONFIG_LWIP_TCP_SND_BUF_DEFAULT** This parameter represents the TX buffer size of the LwIP layer for each TCP stream. Its value should be configured to the value of **WIFI_DYNAMIC_TX_BUFFER_NUM (KB)** to reach a high and stable performance. In case of multiple streams, this value needs to be reduced proportionally.

Throughput optimization by placing code in IRAM:
Section 4. API Guides

- **CONFIG_ESP_WIFI_IRAM_OPT** If this option is enabled, some Wi-Fi functions are moved to IRAM, improving throughput. This increases IRAM usage by 15 kB.
- **CONFIG_ESP_WIFI_RX_IRAM_OPT** If this option is enabled, some Wi-Fi RX functions are moved to IRAM, improving throughput. This increases IRAM usage by 16 kB.
- **CONFIG_LWIP_IRAM_OPTIMIZATION** If this option is enabled, some LwIP functions are moved to IRAM, improving throughput. This increases IRAM usage by 13 kB.

**CACHE:**

- **CONFIG_ESP32S2_INSTRUCTION_CACHE_SIZE** Configure the size of the instruction Cache.
- **CONFIG_ESP32S2_INSTRUCTION_CACHE_LINE_SIZE** Configure the width of the instruction Cache bus.

**Note:** The buffer size mentioned above is fixed as 1.6 KB.

### How to Configure Parameters

The memory of ESP32-S2 is shared by protocol stack and applications.

Here, several configuration ranks are given. In most cases, the user should select a suitable rank for parameter configuration according to the size of the memory occupied by the application.

The parameters not mentioned in the following table should be set to the default.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Iperv</th>
<th>High-performance</th>
<th>Default</th>
<th>Memory saving</th>
<th>Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available memory (KB)</td>
<td>4.1</td>
<td>24.2</td>
<td>78.4</td>
<td>86.5</td>
<td>116.4</td>
</tr>
<tr>
<td>WIFI_STATIC_RX_BUFFER_NUM</td>
<td>16</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>WIFI_DYNAMIC_RX_BUFFER_NUM</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIFI_DYNAMIC_TX_BUFFER_NUM</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIFI_RX_BA_WIN</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>TCP_SND_BUF_DEFAULT</td>
<td>DEFAULT</td>
<td>18</td>
<td>12</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>TCP_WND_DEFAULT (KB)</td>
<td>DEFAULT</td>
<td>18</td>
<td>12</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>WIFI_IRAM_OPT</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>WIFI_RX_IRAM_OPT</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>LWIP_IRAM_OPTIMIZATION</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>INSTRUCTION_CACHE</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>INSTRUCTION_CACHE_LINE</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>TCP throughput (Mbit/s)</td>
<td>37.6</td>
<td>33.1</td>
<td>22.5</td>
<td>12.2</td>
<td>5.5</td>
</tr>
<tr>
<td>TCP throughput (Mbit/s)</td>
<td>31.5</td>
<td>28.1</td>
<td>20.1</td>
<td>13.1</td>
<td>7.2</td>
</tr>
<tr>
<td>UDP throughput (Mbit/s)</td>
<td>58.1</td>
<td>57.3</td>
<td>28.1</td>
<td>22.6</td>
<td>8.7</td>
</tr>
<tr>
<td>UDP throughput (Mbit/s)</td>
<td>78.1</td>
<td>66.7</td>
<td>65.3</td>
<td>53.8</td>
<td>28.5</td>
</tr>
</tbody>
</table>
**Note:** The test was performed with a single stream in a shielded box using an ASUS RT-N66U router. ESP32-S2’s CPU is single core with 240 MHz. ESP32-S2’s flash is in QIO mode with 80 MHz.

**Ranks:**

- **Iperf rank** ESP32-S2 extreme performance rank used to test extreme performance.
- **High-performance rank** The ESP32-S2’s high-performance configuration rank, suitable for scenarios where the application occupies less memory and has high-performance requirements.
- **Default rank** ESP32-S2’s default configuration rank, the available memory, and performance are in balance.
- **Memory saving rank** This rank is suitable for scenarios where the application requires a large amount of memory, and the transceiver performance will be reduced in this rank.
- **Minimum rank** This is the minimum configuration rank of ESP32-S2. The protocol stack only uses the necessary memory for running. It is suitable for scenarios where there is no requirement for performance and the application requires lots of space.

**Using PSRAM**

PSRAM is generally used when the application takes up a lot of memory. In this mode, the `CONFIG_ESP_WIFI_TX_BUFFER` is forced to be static. `CONFIG_ESP_WIFI_STATIC_TX_BUFFER_NUM` indicates the number of DMA buffers at the hardware layer, and increasing this parameter can improve performance. The following are the recommended ranks for using PSRAM:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Iperf</th>
<th>Default</th>
<th>Memory saving</th>
<th>Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available memory (KB)</td>
<td>70.6</td>
<td>96.4</td>
<td>118.8</td>
<td>148.2</td>
</tr>
<tr>
<td>WIFI_STATIC_RX_BUFFER_NUM</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>WIFI_DYNAMIC_RX_BUFFER_NUM</td>
<td>164</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>WIFI_STATIC_TX_BUFFER_NUM</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>WIFI_RX_BA_WIN</td>
<td>6</td>
<td>6</td>
<td>Disable</td>
<td></td>
</tr>
<tr>
<td>TCP_SND_BUF_DEFAULT (KB)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>TCP_WND_DEFAULT (KB)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>WIFI_IRAM_OPT</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>WIFI_RX_IRAM_OPT</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>LWIP_IRAM_OPT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>INSTRUCTION_CACHE</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>INSTRUCTION_CACHE_LINE</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>DATA_CACHE</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>DATA_CACHE_LINE</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TCPTX throughput (Mbit/s)</td>
<td>40.1</td>
<td>29.2</td>
<td>20.1</td>
<td>8.9</td>
</tr>
<tr>
<td>TCP RX throughput (Mbit/s)</td>
<td>21.9</td>
<td>16.8</td>
<td>14.8</td>
<td>9.6</td>
</tr>
<tr>
<td>UDP throughput (Mbit/s)</td>
<td>50.1</td>
<td>25.7</td>
<td>22.4</td>
<td>10.2</td>
</tr>
<tr>
<td>UDP throughput (Mbit/s)</td>
<td>45.3</td>
<td>43.1</td>
<td>28.5</td>
<td>15.1</td>
</tr>
</tbody>
</table>

**Note:** Reaching peak performance may cause task watchdog. It is a normal phenomenon considering the CPU may...
have no time for lower priority tasks.

### 4.31.35 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. 53: TX Buffer Allocation](image)

**Description:**
- The application allocates the data which needs to be sent out.
- The application calls TCPIP-/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:

![Fig. 54: RX Buffer Allocation](image)

**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.

The diagram shows the configuration of the Wi-Fi internal buffer.
<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Alloc Type</th>
<th>Default</th>
<th>Configurable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static RX Buffer (Hardware RX</td>
<td>Static</td>
<td>10 * 1600 Bytes</td>
<td>Yes</td>
<td>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. If needs be, the application can reduce the memory statically allocated by Wi-Fi. It can reduce this value from 10 to 6 to save 6400 Bytes of memory. It is not recommended to reduce the configuration to a value less than 6 unless the AMPDU feature is disabled.</td>
</tr>
<tr>
<td>Buffer)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic RX Buffer</td>
<td>Dynamic</td>
<td>32</td>
<td>Yes</td>
<td>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 menu-config, is used to limit the total un-freed Dynamic Rx Buffer number.</td>
</tr>
<tr>
<td>Dynamic TX Buffer</td>
<td>Dynamic</td>
<td>32</td>
<td>Yes</td>
<td>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 a ‘Dynamic TX Buffer’ and makes a copy of the upper-layer buffer. The Dynamic and Static TX Buffers are mutually exclusive.</td>
</tr>
<tr>
<td>Static TX Buffer</td>
<td>Static</td>
<td>16 * 1600 Bytes</td>
<td>Yes</td>
<td>This is a kind of DMA memory. It is initialized in <code>esp_wifi_init()</code> and freed in <code>esp_wifi_deinit()</code>. When the upper-layer (LwIP) sends packets to the Wi-Fi driver, it firstly allocates a ‘Static TX Buffer’ and makes a copy of the upper-layer buffer. The Dynamic and Static TX Buffer are mutually exclusive.</td>
</tr>
<tr>
<td>Management Short Buffer</td>
<td>Dynamic</td>
<td>8</td>
<td>NO</td>
<td>Wi-Fi driver’s internal buffer.</td>
</tr>
<tr>
<td>Management Long Buffer</td>
<td>Dynamic</td>
<td>32</td>
<td>NO</td>
<td>Wi-Fi driver’s internal buffer.</td>
</tr>
<tr>
<td>Management Long Long Buffer</td>
<td>Dynamic</td>
<td>32</td>
<td>NO</td>
<td>Wi-Fi driver’s internal buffer.</td>
</tr>
</tbody>
</table>
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 the 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 Aggregate MAC Protocol Data Unit (AMPDU)

ESP32-S2 supports both receiving and transmitting AMPDU, and the AMPDU can greatly improve the Wi-Fi throughput.

Generally, the AMPDU should be enabled. Disabling AMPDU is usually for debugging purposes.

4.31.36 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 Cannot 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.

1. Where to Get Wireshark

You can get Wireshark from the official website: [https://www.wireshark.org/download.html](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.


**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.

```bash
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

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.
Chapter 4. API Guides

Fig. 55: Wireshark Capture Interface

Fig. 56: Setting up Filters in Wireshark
Chapter 4. API Guides

Click Filter, the top left blue button in the picture below. The display filter dialogue box will appear.

```
<table>
<thead>
<tr>
<th></th>
<th>apply a display filter -- &lt;filter&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage display filters</td>
<td>manage filter expressions</td>
</tr>
<tr>
<td>non-http and non-smtp to/from 192.0.2.1: ip.addr == 192.0.2.1 and not tcp.port in (80, 25)</td>
<td></td>
</tr>
<tr>
<td>no arp and no dns: not arp and !udp.port == 53</td>
<td></td>
</tr>
<tr>
<td>http: http</td>
<td>tcp or udp port is 80 (http): tcp.port == 80</td>
</tr>
<tr>
<td>non-dns: !udp.port == 53</td>
<td>!tcp.port == 53</td>
</tr>
<tr>
<td>udp only: udp</td>
<td>tcp only: tcp</td>
</tr>
<tr>
<td>ipx only: ipx</td>
<td>ipv6 address 2001:db8::1: ipv6.addr == 2001:db8::1</td>
</tr>
<tr>
<td>ipv6 only: ipv6</td>
<td>ipv4 address isn't 192.0.2.1 (don't use ! for this): !ipv4.addr == 192.0.2.1</td>
</tr>
<tr>
<td>ipv4 address 192.0.2.1: ipv4.addr == 192.0.2.1</td>
<td>ipv4 only: ip</td>
</tr>
<tr>
<td>no arp: not arp</td>
<td>ethernet broadcast: eth.addr == broadcast address</td>
</tr>
<tr>
<td>ethernet type 0x0806 (arp): eth.type == 0x0806</td>
<td>ethernet address 00:00:5e:00:53:00: eth.addr == 00:00:5e:00:53:00</td>
</tr>
</tbody>
</table>
```

Fig. 57: 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.

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.

f) Stop/Start Packet Capture

As shown in the picture below, click the red button to stop capturing the current packet.

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.

### 4.32 Wi-Fi Security

#### 4.32.1 ESP32-S2 Wi-Fi Security Features

- Support for Protected Management Frames (PMF)
- Support for WPA3-Personal
Fig. 58: Filter Expression Dialogue Box

Fig. 59: Filter Toolbar

Fig. 60: Example of MAC Addresses applied in the Filter Toolbar
Fig. 61: Example of Packet List Details

Fig. 62: Stopping Packet Capture

Fig. 63: Starting or Resuming the Packets Capture
Support for Opportunistic Wireless Encryption (OWE)

In addition to traditional security methods (WEP/WPA-TKIP/WPA2-CCMP), ESP32-S2 Wi-Fi supports state-of-the-art security protocols, namely Protected Management Frames (PMF), Wi-Fi Protected Access 3 and Enhanced Open™ based on Opportunistic Wireless Encryption. WPA3 provides better privacy and robustness against known attacks on traditional modes. Enhanced Open™ enhances the security and privacy of users connecting to open (public) Wireless Networks without authentication.

### 4.3.2.2 Protected Management Frames (PMF)

**Introduction**

In Wi-Fi, management frames such as beacons, probes, authentication/deauthentication, and association/disassociation 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 authentication/deauthentication or association/disassociation frames at the right time, leading to attacks such as Denial-of-Service (DOS) and man-in-the-middle.

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 a Secure Association (SA) teardown mechanism to prevent spoofed association/authentication frames from disconnecting already connected clients.

There are three types of PMF configuration modes on both the station and AP sides:

- PMF Optional
- PMF Required
- PMF Disabled

**API & Usage**

ESP32-S2 supports PMF in both the station and SoftAP mode. For both, the default mode is PMF Optional. For even higher security, PMF Required mode can be enabled by setting the `required` flag in `pmf_cfg` while using the `esp_wifi_set_config()` API. This results in the device only connecting to a PMF-enabled device and...
rejecting others. PMF Optional can be disabled using `esp_wifi_disable_pmf_config()` API. If SoftAP is started in WPA3 or WPA2/WPA3 mixed mode, trying to disable PMF results in an error.

**Attention:** capable flag in pmf_cfg is deprecated and set to true internally. This is to take the additional security benefit of PMF whenever possible.

### 4.32.3 Wi-Fi Enterprise

**Introduction**

Enterprise security is the secure authentication mechanism for enterprise wireless networks. It uses the RADIUS server for authentication of network users before connecting to the Access Point (AP). The authentication process is based on 802.1X policy and comes with different Extended Authentication Protocol (EAP) methods such as TLS, TTLS, PEAP, and EAP-FAST. RADIUS server authenticates the users based on their credentials (username and password), digital certificates, or both.

**Note:** ESP32-S2 supports Wi-Fi Enterprise only in station mode.

ESP32-S2 supports **WPA2-Enterprise** and **WPA3-Enterprise**. WPA3-Enterprise builds upon the foundation of WPA2-Enterprise with the additional requirement of using Protected Management Frames (PMF) and server certificate validation on all WPA3 connections. **WPA3-Enterprise also offers an additional secure mode using 192-bit minimum-strength security protocols and cryptographic tools to better protect sensitive data.** The 192-bit security mode offered by WPA3-Enterprise ensures the right combination of cryptographic tools is used and sets a consistent baseline of security within a WPA3 network. WPA3-Enterprise 192-bit mode is only supported by modules having `SOC_WIFI_GCMP_SUPPORT` support. Enable `CONFIG_ESP_WIFI_SUITE_B_192` flag to support WPA3-Enterprise with 192-bit mode.

**ESP32-S2 supports the following EAP methods:**

- **EAP-TLS:** This is a certificate-based method and only requires SSID and EAP-IDF.
- **PEAP:** This is a Protected EAP method. Usernames and passwords are mandatory.
- **EAP-TTLS:** This is a credential-based method. Only server authentication is mandatory while user authentication is optional. Username and Password are mandatory. It supports different Phase2 methods, such as:
  - MSCHAP and MSCHAP-V2.
- **EAP-FAST:** This is an authentication method based on Protected Access Credentials (PAC) which also uses identity and password. Currently, `CONFIG_ESP_WIFI_MBEDTLS_TLS_CLIENT` flag should be disabled to use this feature.

Example `wifi/wifi_enterprise` demonstrates all the supported Wi-Fi Enterprise methods except EAP-FAST. Please refer `wifi/wifi_eap_fast` for the EAP-FAST example. EAP method can be selected from the Example Configuration menu in `idf.py menuconfig`. Refer to `examples/wifi/wifi_enterprise/README.md` for information on how to generate certificates and run the example.

### 4.32.4 WPA3-Personal

**Introduction**

Wi-Fi Protected Access-3 (WPA3) is a set of enhancements to Wi-Fi access security intended to replace the current WPA2 standard. It includes new features and capabilities that offer significantly better protection against different types of attacks. It improves upon WPA2-Personal in the following ways:

- WPA3 uses Simultaneous Authentication of Equals (SAE), which is a password-authenticated key agreement method based on Diffie-Hellman key exchange. Unlike WPA2, the technology is resistant to offline-dictionary
attacks, where the attacker attempts to determine a shared password based on a captured 4-way handshake without any further network interaction.

- Disallows outdated protocols such as TKIP, which is susceptible to simple attacks like MIC key recovery attacks.
- Mandates Protected Management Frames (PMF), which provides protection for unicast and multicast robust management frames which include Disassoc and Deauth frames. This means that the attacker cannot disrupt an established WPA3 session by sending forged Assoc frames to the AP or Deauth/Disassoc frames to the station.
- Provides forward secrecy, which means the captured data cannot be decrypted even if the password is compromised after data transmission.

ESP32-S2 station also supports following additional Wi-Fi CERTIFIED WPA3™ features:

- **Transition Disable**: WPA3 defines transition modes for client devices so that they can connect to a network even when some of the APs in that network do not support the strongest security mode. Client device implementations typically configure network profiles in a transition mode by default. However, such a client device could be subject to an active downgrade attack in which the attacker causes the client device to use a lower security mode in order to exploit a vulnerability with that mode. WPA3 has introduced the Transition Disable feature to mitigate such attacks, by enabling client devices to change from a transition mode to an “only” mode when connecting to a network, once that network indicates it fully supports the higher security mode. Enable `transition_disable` in `wifi_sta_config_t` to enable this feature for ESP32-S2 station.

- **SAE PUBLIC-KEY (PK)**: As the password at small public networks is shared with multiple users, it may be relatively easy for an attacker to find out the password, which is sufficient to launch an evil twin attack. Such attacks are prevented by an extension to WPA3-Personal called SAE-PK. The SAE-PK authentication exchange is very similar to the regular SAE exchange, with the addition of a digital signature sent by the AP to the client device. The client device validates the public key asserted by the AP based on the password fingerprint and verifies the signature using the public key. So even if the attacker knows the password, it does not know the private key to generate a valid signature, and therefore the client device is protected against an evil twin attack. Enable `CONFIG_ESP_WIFI_ENABLE_SAE_PK` and `sae_pk_mode` in `wifi_sta_config_t` to add support of SAE PK for ESP32-S2 station.

- **SAEPWEMethods**: ESP32-S2 station as well as SoftAP supports SAE Password Element derivation method Hunting And Pecking and Hash to Element (H2E). H2E is computationally efficient as it uses fewer iterations than Hunt and Peck, and also it mitigates side-channel attacks. These can be configured using the parameter `sae_pwe_h2e` from `wifi_sta_config_t` and `wifi_ap_config_t` for station and SoftAP respectively. Hunt and Peck, H2E both can be enabled by using `WPA3_SAE_PWE_BOTH` configuration.

Please refer to the Security section of Wi-Fi Alliance’s official website for further details.

**Setting up WPA3 Personal with ESP32-S2**

A configuration option `CONFIG_ESP_WIFI_ENABLE_WPA3_SAE` is provided to enable/disable WPA3 for the station. By default, it is kept enabled. If disabled, ESP32-S2 will not be able to establish a WPA3 connection. Also under the Wi-Fi component, a configuration option `CONFIG_ESP_WIFI_SOFTAP_SAE_SUPPORT` is provided to enable/disable WPA3 for SoftAP. Additionally, since PMF is mandated by WPA3 protocol, PMF Optional is set by default for station and SoftAP. PMF Required can be configured using Wi-Fi configuration. For WPA3 SoftAP, PMF Required is mandatory and will be configured and stored in NVS implicitly if not specified by the user.

Refer to Protected Management Frames (PMF) on how to set this mode.

After configuring all required settings for the WPA3-Personal station, application developers need not worry about the underlying security mode of the AP. WPA3-Personal is now the highest supported protocol in terms of security, so it is automatically selected for the connection whenever available. For example, if an AP is configured to be in WPA3 Transition Mode, where it advertises as both WPA2 and WPA3 capable, the station chooses WPA3 for the connection with the above settings.

After configuring all required setting for WPA3-Personal SoftAP, application developers have to set `WIFI_AUTH_WPA3_PSK` for `authmode` in `wifi_ap_config_t` to start AP in WPA3 security. SoftAP can be also configured to use `WIFI_AUTH_WPA2_WPA3_PSK` mixed mode.

Note that binary size will be increased by about 6.5 kilobytes after enabling `CONFIG_ESP_WIFI_SOFTAP_SAE_SUPPORT`. 
4.32.5 Wi-Fi Enhanced Open™

Introduction

Enhanced Open™ is used for providing security and privacy to users connecting to open (public) wireless networks, particularly in scenarios where user authentication is not desired or distribution of credentials impractical. Each user is provided with unique individual encryption keys that protect data exchange between a user device and the Wi-Fi network. Protected Management Frames further protect management traffic between the access point and the user device. Enhanced Open™ is based on the Opportunistic Wireless Encryption (OWE) standard. OWE Transition Mode enables a seamless transition from Open unencrypted WLANs to OWE WLANs without adversely impacting the end-user experience.

Note: ESP32-S2 supports Wi-Fi Enhanced Open™ only in station mode.

Setting up OWE with ESP32-S2

A configuration option `CONFIG_ESP_WIFI_ENABLE_WPA3_OWE_STA` and configuration parameter `owe_enabled` in `wifi_sta_config_t` is provided to enable OWE support for the station. To use OWE transition mode, along with the configuration provided above, `authmode` from `wifi_scan_threshold_t` should be set to `WIFI_AUTH_OPEN`. 
Chapter 5

Security Guides

5.1 Overview

5.1.1 Security

This guide provides an overview of the overall security features available in various Espressif solutions. It is highly recommended to consider this guide while designing the products with the Espressif platform and the ESP-IDF software stack from the security perspective.

Goals

High level security goals are as follows:

1. Preventing untrustworthy code from being executed
2. Protecting the identity and integrity of the code stored in the off-chip flash memory
3. Securing device identity
4. Secure storage for confidential data
5. Authenticated and encrypted communication from the device

Platform Security

Secure Boot    The Secure Boot feature ensures that only authenticated software can execute on the device. The Secure Boot process forms a chain of trust by verifying all mutable software entities involved in the Application Startup Flow. Signature verification happens during both boot-up as well as in OTA updates.

Please refer to Secure Boot V2 for detailed documentation about this feature.

Important:  It is highly recommended that Secure Boot be enabled on all production devices.

Secure Boot Best Practices

- Generate the signing key on a system with a quality source of entropy.
- Always keep the signing key private. 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.
• Ensure that all security eFuses have been correctly programmed, including disabling of the debug interfaces, non-required boot mediums (e.g., UART DL mode), etc.

Flash Encryption  The Flash Encryption feature helps to encrypt the contents on the off-chip flash memory and thus provides the confidentiality aspect to the software or data stored in the flash memory.

Please refer to Flash Encryption for detailed information about this feature.

If ESP32-S2 is connected to an external SPI RAM, the contents written to or read from the SPI RAM will also be encrypted and decrypted respectively (via the MMU’s flash cache, provided that Flash Encryption is enabled). This provides an additional safety layer for the data stored in SPI RAM, hence configurations like `CONFIG_MBEDTLS_EXTERNAL_MEM_ALLOC` can be safely enabled in this case.

Flash Encryption Best Practices
• It is recommended to use flash Encryption release mode for the production use-cases.
• It is recommended to have a unique flash encryption key per device.
• Enable Secure Boot as an extra layer of protection, and to prevent an attacker from selectively corrupting any part of the flash before boot.

Device Identity  The Digital Signature peripheral in ESP32-S2 produces hardware-accelerated RSA digital signatures with the assistance of HMAC, without the RSA private key being accessible by software. This allows the private key to be kept secured on the device without anyone other than the device hardware being able to access it.

This peripheral can help to establish the Secure Device Identity to the remote endpoint, e.g., in the case of TLS mutual authentication based on the RSA cipher scheme.

Please refer to the Digital Signature (DS) for detailed documentation.

Memory Protection  ESP32-S2 supports the Memory Protection scheme, either through architecture or special peripheral like PMS, which provides an ability to enforce and monitor permission attributes to memory and, in some cases, peripherals. ESP-IDF application startup code configures the permissions attributes like Read/Write access on data memories and Read/Execute access on instruction memories using the relevant peripheral. If there is any attempt made that breaks these permission attributes, e.g., a write operation to instruction memory region, then a violation interrupt is raised, and it results in system panic.

This feature depends on the config option `CONFIG_ESP_SYSTEM_MEMPROT_FEATURE` and it is kept enabled by default. Please note that the API for this feature is private and used exclusively by ESP-IDF code only.

Note: This feature can help to prevent the possibility of remote code injection due to the existing vulnerabilities in the software.

Debug Interfaces

JTAG
• JTAG interface stays disabled if any of the security features are enabled. Please refer to JTAG with Flash Encryption or Secure Boot for more information.
• JTAG interface can also be disabled in the absence of any other security features using eFuse API.
• ESP32-S2 supports soft disabling the JTAG interface and it can be re-enabled by programming a secret key through HMAC. (HMAC for Enabling JTAG)
**UART Download Mode**  In ESP32-S2, Secure UART Download mode gets activated if any of the security features are enabled.

- Secure UART Download mode can also be enabled by calling `esp_efuse_enable_rom_secure_download_mode()`.
- This mode does not allow any arbitrary code to execute if downloaded through the UART download mode.
- It also limits the available commands in Download mode to update SPI config, e.g., changing baud rate, basic flash write, and the command to return a summary of currently enabled security features (`get_security_info`).
- To disable Download Mode entirely, select the `CONFIG_SECURE_UART_ROM_DL_MODE` to the recommended option Permanently disable ROM Download Mode or call `esp_efuse_disable_rom_download_mode()` at runtime.

**Important:** In Secure UART Download mode, esptool.py can only work with the argument `--no-stub`.

**Network Security**

**Wi-Fi**  In addition to the traditional security methods (WEP/WPA-TKIP/WPA2-CCMP), Wi-Fi driver in ESP-IDF also supports additional state-of-the-art security protocols. Please refer to the Wi-Fi Security for detailed documentation.

**TLS (Transport Layer Security)**  It is recommended to use TLS (Transport Layer Security) in all external communications (e.g., cloud communication, OTA updates) from the ESP device. ESP-IDF supports Mbed TLS as the official TLS stack.

TLS is default integrated in ESP HTTP Client, HTTPS Server and several other components that ship with ESP-IDF.

**Note:** It is recommended to use the ESP-IDF protocol components in their default configuration, which has been ensured to be secure. Disabling of HTTPS and similar security-critical configurations should be avoided.

**ESP-TLS Abstraction**  ESP-IDF provides an abstraction layer for the most-used TLS functionalities. Hence, it is recommended that an application uses the API exposed by ESP-TLS.

**TLS Server Verification** section highlights diverse ways in which the identity of server could be established on the device side.

**ESP Certificate Bundle**  The ESP x509 Certificate Bundle API provides an easy way to include a bundle of custom x509 root certificates for TLS server verification. The certificate bundle is the easiest way to verify the identity of almost all standard TLS servers.

**Important:** It is highly recommended to verify the identity of the server based on X.509 certificates to avoid establishing communication with the fake server.

**Managing Root Certificates**  Root Certificates embedded inside the application must be managed carefully. Any update to the root certificate list or the ESP x509 Certificate Bundle can have an impact on the TLS connection with the remote endpoint. This includes a connection to the OTA update server. In some cases, the problem shall be visible on the next OTA update and it may leave device unable to perform OTA updates forever.

Root certificates list update could have following reasons:

- New firmware has different set of remote endpoint(s).
- The existing certificate has expired.
- The certificate has been added or retracted from the upstream certificate bundle.
Chapter 5. Security Guides

- The certificate list changed due to market share statistics (CONF_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_CMN case).

Some guidelines to consider on this topic:

- Please consider enabling OTA rollback and then keep the successful connection to the OTA update server as the checkpoint to cancel the rollback process. This ensures that the newly updated firmware can successfully reach till the OTA update server, otherwise rollback process will go back to the previous firmware on the device.
- If you plan to enable the CONFIG_MBEDTLS_HAVE_TIME_DATE option, then please consider to have the time sync mechanism (SNTP) and sufficient number of trusted certificates in place.

Product Security

Secure Provisioning Secure Provisioning refers to a process of secure on-boarding of the ESP device on to the Wi-Fi network. This mechanism also allows provision of additional custom configuration data during the initial provisioning phase from the provisioning entity, e.g., Smartphone.

ESP-IDF provides various security schemes to establish a secure session from the ESP and the provisioning entity, they are highlighted at Security Schemes.

Please refer to the Wi-Fi Provisioning documentation for details and the example code for this feature.

Note: Espressif provides Android and iOS Phone Apps along with their sources, so that it could be easy to further customize them as per the product requirement.

Secure OTA (Over-the-air) Updates

- OTA Updates must happen over secure transport, e.g., HTTPS.
- ESP-IDF provides a simplified abstraction layer ESP HTTPS OTA for this.
- If Secure Boot is enabled, then the server should host the signed application image.
- If Flash Encryption is enabled, then no additional steps are required on the server side, encryption shall be taken care on the device itself during flash write.
- OTA update Rollback Process can help to switch the application as active only after its functionality has been verified.

Anti-Rollback Protection Anti-rollback protection feature ensures that device only executes the application that meets the security version criteria as stored in its eFuse. So even though the application is trusted and signed by legitimate key, it may contain some revoked security feature or credential. Hence, device must reject any such application.

ESP-IDF allows this feature for the application only and it is managed through 2nd stage bootloader. The security version is stored in the device eFuse and it is compared against the application image header during both bootup and over-the-air updates.

Please see more information to enable this feature in the Anti-rollback guide.

Encrypted Firmware Distribution Encrypted firmware distribution during over-the-air updates ensures that the application stays encrypted in transit from the server to the device. This can act as an additional layer of protection on top of the TLS communication during OTA updates and protect the identity of the application.

Please see working example for this documented in OTA Upgrades with Pre-Encrypted Firmware section.

Secure Storage Secure storage refers to the application-specific data that can be stored in a secure manner on the device, i.e., off-chip flash memory. This is typically a read-write flash partition and holds device specific configuration data, e.g., Wi-Fi credentials.

ESP-IDF provides the NVS (Non-volatile Storage) management component which allows encrypted data partitions. This feature is tied with the platform Flash Encryption feature described earlier.
Please refer to the *NVS Encryption* for detailed documentation on the working and instructions to enable this feature.

**Important:** By default, ESP-IDF components writes the device specific data into the default NVS partition, including Wi-Fi credentials too, and it is recommended to protect this data using *NVS Encryption* feature.

---

**Secure Device Control** ESP-IDF provides capability to control an ESP device over *Wi-Fi + HTTP* or *BLE* in a secure manner using ESP Local Control component. 

Please refer to the *ESP Local Control* for detailed documentation about this feature.

---

**Security Policy**

The ESP-IDF GitHub repository has attached *Security Policy Brief*.

---

**Advisories**

- Espressif publishes critical *Security Advisories*, which includes security advisories regarding both hardware and software.
- The specific advisories of the ESP-IDF software components are published through the *GitHub repository*.

---

**Software Updates** Critical security issues in the ESP-IDF components, and third-party libraries are fixed as and when we find them or when they are reported to us. Gradually, we make the fixes available in all applicable release branches in ESP-IDF.

Applicable security issues and CVEs for the ESP-IDF components, third-party libraries are mentioned in the ESP-IDF release notes.

**Important:** We recommend periodically updating to the latest bugfix version of the ESP-IDF release to have all critical security fixes available.

---

### 5.2 Features

#### 5.2.1 Flash Encryption

This is a quick start guide to ESP32-S2’s flash encryption feature. Using application code as an example, it demonstrates how to test and verify flash encryption operations during development and production.

**Introduction**

Flash encryption is intended for encrypting the contents of the ESP32-S2’s off-chip flash memory. Once this feature is enabled, firmware is flashed as plaintext, and then the data is encrypted in place on the first boot. As a result, physical readout of flash will not be sufficient to recover most flash contents.

**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 ESP32-S2. Before using this feature, read the document and make sure to understand the implications.
Encrypted Partitions

With flash encryption enabled, the following types of data are encrypted by default:

- *Second Stage Bootloader* (Firmware Bootloader)
- Partition Table
- *NVS Key Partition*
- Otadata
- All app type partitions

Other types of data can be encrypted conditionally:

- Any partition marked with the encrypted flag in the partition table. For details, see *Encrypted Partition Flag*.
- Secure Boot bootloader digest if Secure Boot is enabled (see below).

Relevant eFuses

The flash encryption operation is controlled by various eFuses available on ESP32-S2. The list of eFuses and their descriptions is given in the table below. The names in eFuse column are also used by esefuse.py tool. For usage in the eFuse API, modify the name by adding ESP_EFUSE_, for example: esp_efuse_read_field_bit(ESP_EFUSE_DISABLE_DL_ENCRYPT).

<table>
<thead>
<tr>
<th>eFuse</th>
<th>Description</th>
<th>Bit Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK_KEYN</td>
<td>AES key storage. N is between 0 and 5.</td>
<td>One 256 bit key block for XTS_AES_128, Two 256 bit key blocks for XTS_AES_256 (512 bit total)</td>
</tr>
<tr>
<td>KEY_PURPOSE_N</td>
<td>Controls the purpose of eFuse block BLOCK_KEYN, where N is between 0 and 5. Possible values: 2 for XTS_AES_256_KEY_1, 3 for XTS_AES_256_KEY_2, and 4 for XTS_AES_128_KEY. Final AES key is derived based on the value of one or two of these purpose eFuses. For a detailed description of the possible combinations, see ESP32-S2 Technical Reference Manual &gt; External Memory Encryption and Decryption (XTS_AES) [PDF].</td>
<td>4</td>
</tr>
<tr>
<td>DIS_DOWNLOAD_MANUAL_ENCRYPT</td>
<td>If set, disables flash encryption when in download boot-modes.</td>
<td>1</td>
</tr>
<tr>
<td>SPI_BOOT_CRYPT_CNT</td>
<td>Enables encryption and decryption, when an SPI boot mode is set. Feature is enabled if 1 or 3 bits are set in the eFuse, disabled otherwise.</td>
<td>3</td>
</tr>
</tbody>
</table>

Note:

- R/W access control is available for all the eFuse bits listed in the table above.
- The default value of these bits is 0 after manufacturing.

Read and write access to eFuse bits is controlled by appropriate fields in the registers WR_DIS and RD_DIS. For more information on ESP32-S2 eFuses, see *eFuse manager*. To change protection bits of eFuse field using...
espefuse.py, use these two commands: read_protect_efuse and write_protect_efuse. Example espefuse.py write_protect_efuse DISABLE_DL_ENCRYPT.

Flash Encryption Process

Assuming that the eFuse values are in their default states and the firmware bootloader is compiled to support flash encryption, the flash encryption process executes as shown below:

1. On the first power-on reset, all data in flash is un-encrypted (plaintext). The ROM bootloader loads the firmware bootloader.
2. Firmware bootloader reads the SPI_BOOT_CRYPT_CNT eFuse value (0b000). Since the value is 0 (even number of bits set), it configures and enables the flash encryption block. For more information on the flash encryption block, see ESP32-S2 Technical Reference Manual > eFuse Controller (eFuse) > Auto Encryption Block [PDF].
3. Firmware bootloader first checks if a valid key is already present in the eFuse (e.g., burned using espefuse tool) then the process of key generation is skipped and the same key is used for flash encryption process. Otherwise, Firmware bootloader uses RNG (random) module to generate an 256 bit or 512 bit key, depending on the value of Size of generated XTS-AES key, and then writes it into respectively one or two BLOCK_KEY eFuses. The software also updates the KEY_PURPOSE_N for the blocks where the keys were stored. The key cannot be accessed via software as the write and read protection bits for one or two BLOCK_KEY eFuses are set. KEY_PURPOSE_N field is write-protected as well. The flash encryption operations happen entirely by hardware, and the key cannot be accessed via software.
4. Flash encryption block encrypts the flash contents - the firmware bootloader, applications and partitions marked as encrypted. Encrypting in-place can take time, up to a minute for large partitions.
5. Firmware bootloader sets the first available bit in SPI_BOOT_CRYPT_CNT (0b001) to mark the flash contents as encrypted. Odd number of bits is set.
6. For Development Mode, the firmware bootloader allows the UART bootloader to re-flash encrypted binaries. Also, the SPI_BOOT_CRYPT_CNT eFuse bits are NOT write-protected. In addition, the firmware bootloader by default sets the following eFuse bits:

   - DIS_BOOT_REMAP
   - DIS_DOWNLOAD_ICACHE
   - DIS_DOWNLOAD_DCACHE
   - HARD_DIS_JTAG
   - DIS_LEGACY_SPI_BOOT

7. For Release Mode, the firmware bootloader sets all the eFuse bits set under development mode as well as DIS_DOWNLOAD_MANUAL_ENCRYPT. It also write-protection the SPI_BOOT_CRYPT_CNT eFuse bits. To modify this behavior, see Enabling UART Bootloader Encryption/Decryption.
8. The device is then rebooted to start executing the encrypted image. The firmware bootloader calls the flash decryption block to decrypt the flash contents and then loads the decrypted contents into IRAM.

During the development stage, there is a frequent need to program different plaintext flash images and test the flash encryption process. This requires that Firmware Download mode is able to load new plaintext images as many times as it might be needed. However, during manufacturing or production stages, Firmware Download mode should not be allowed to access flash contents for security reasons.

Hence, two different flash encryption configurations were created: for development and for production. For details on these configurations, see Section Flash Encryption Configuration.

Flash Encryption Configuration

The following flash encryption modes are available:

- Development Mode - recommended for use only during development. In this mode, it is still possible to flash new plaintext firmware to the device, and the bootloader will transparently encrypt this firmware using the key stored in hardware. This allows, indirectly, to read out the plaintext of the firmware in flash.
Chapter 5. Security Guides

- **Release Mode** - recommended for manufacturing and production. In this mode, flashing plaintext firmware to the device without knowing the encryption key is no longer possible.

This section provides information on the mentioned flash encryption modes and step by step instructions on how to use them.

**Development Mode**  
During development, you can encrypt flash using either an ESP32-S2 generated key or external host-generated key.

**Using ESP32-S2 Generated Key**  
Development mode allows you to download multiple plaintext images using Firmware Download mode.

To test flash encryption process, take the following steps:

1. Ensure that you have an ESP32-S2 device with default flash encryption eFuse settings as shown in Relevant eFuses.
   
   See how to check ESP32-S2 Flash Encryption Status.

2. In Project Configuration Menu, do the following:
   
   - Enable flash encryption on boot.
   - Select encryption mode (Development mode by default).
   - Select UART ROM download mode (enabled by default).
   - Set Size of generated XTS-AES key.
   - Select the appropriate bootloader log verbosity.
   - Save the configuration and exit.

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See Bootloader Size.

3. Run the command given below to build and flash the complete images.

   ```
   idf.py flash monitor
   ```

   **Note:** This command does not include any user files which should be written to the partitions on the flash memory. Please write them manually before running this command otherwise the files should be encrypted separately before writing.

This command will write to flash memory unencrypted images: the firmware bootloader, the partition table and applications. Once the flashing is complete, ESP32-S2 will reset. On the next boot, the firmware bootloader encrypts: the firmware bootloader, application partitions and partitions marked as encrypted then resets. Encrypting in-place can take time, up to a minute for large partitions. After that, the application is decrypted at runtime and executed.

A sample output of the first ESP32-S2 boot after enabling flash encryption is given below:

```
ESP-ROM:esp32s2-rc4-20191025
Build:Oct 25 2019
rst:0x1 (POWERON),boot:0x8 (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3ffe6260,len:0x78
load:0x3ffe62d8,len:0x231c
load:0x4004c000,len:0x9d8
load:0x40050000,len:0x3cf8
entry 0x4004c1ec
I (48) boot: ESP-IDF qa-test-v4.3-20201113-777-gd8e1 2nd stage bootloader
I (48) boot: compile time 11:24:04
```
I (48) boot: chip revision: 0
I (52) boot.esp32s2: SPI Speed : 80MHz
I (57) boot.esp32s2: SPI Mode : DIO
I (62) boot.esp32s2: SPI Flash Size : 2MB
I (66) boot: Enabling RNG early entropy source...
I (72) boot: Partition Table:
I (75) boot: ## Label Usage Type ST Offset Length
I (83) boot: 0 nvs WiFi data 01 02 0000a000 00006000
I (90) boot: 1 storage Unknown data 01 ff 00010000 00001000
I (98) boot: 2 factory factory app 00 00 00020000 00100000
I (105) boot: End of partition table
I (109) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f000020 size=0x0618c (~24972) map
I (124) esp_image: segment 1: paddr=0x000261b4 vaddr=0x3ffbcae0 size=0x02624 (~9764) load
I (129) esp_image: segment 2: paddr=0x000287e0 vaddr=0x40022000 size=0x00404 (~1028) load
0x40022000: _WindowOverflow4 at /home/marius/esp-idf/components/freertos/port/xtensa/xtensa_vectors.S:1730
I (136) esp_image: segment 3: paddr=0x00028bce vaddr=0x40022404 size=0x0742c (~29740) load
0x40022404: _coredump_iram_end at ???:
I (153) esp_image: segment 4: paddr=0x00030020 vaddr=0x40080020 size=0x1457c (~83324) map
0x40080020: _stext at ???:
I (171) esp_image: segment 5: paddr=0x000445a4 vaddr=0x40029830 size=0x032ac (~12972) load
0x40029830: gpspi_flash_ll_set_miso_bitlen at /home/marius/esp-idf/examples/security/flash_encryption/build/../../../../components/hal/esp32s2/include/hal/gpspi_flash_ll.h:261 (inlined by) spi_flash_hal_gpspi_common_command at /home/marius/esp-idf/components/hal/spi_flash_hal_common.inc:161
I (181) boot: Loaded app from partition at offset 0x20000
I (181) boot: Checking flash encryption...
I (181) efuse: Batch mode of writing fields is enabled
I (188) flash_encrypt: Generating new flash encryption key...
I (199) flash_encrypt: Not disabling UART bootloader encryption
I (201) flash_encrypt: Disable UART bootloader cache...
I (207) flash_encrypt: Disable JTAG...
I (212) efuse: Batch mode of writing fields is disabled
I (217) esp_image: segment 0: paddr=0x00001020 vaddr=0x3ffe6260 size=0x00078 (~120)
I (226) esp_image: segment 1: paddr=0x000010a0 vaddr=0x3ffe62d8 size=0x0231c (~8988)
I (236) esp_image: segment 2: paddr=0x000033c4 vaddr=0x40040c00 size=0x009d8 (~2520)
I (243) esp_image: segment 3: paddr=0x0000003da4 vaddr=0x40005000 size=0x03cf8 (~15608)
I (651) flash_encrypt: bootloader encrypted successfully
I (704) flash_encrypt: partition table encrypted and loaded successfully
I (704) flash_encrypt: Encrypting partition 1 at offset 0x10000 (length 0x1000)...
I (765) flash_encrypt: Done encrypting
I (766) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f000020 size=0x0618c (~24972) map
I (773) esp_image: segment 1: paddr=0x000261b4 vaddr=0x3ffbcae0 size=0x02624 (~9764)
I (778) esp_image: segment 2: paddr=0x000287e0 vaddr=0x40022000 size=0x00404 (~1028)
(continues on next page)
A sample output of subsequent ESP32-S2 boots just mentions that flash encryption is already enabled:

ESP-ROM:esp32s2-rc4-20191025
Build:Oct 25 2019
rst:0x3 (RTC_SW_SYS_RST),boot:0x8 (SPI_FAST_FLASH_BOOT)
Saved PC:0x40051242
SPIWP:0xee
mode:DIO, clock div:1
load:0x3ffe6260,len:0x78
load:0x3ffe62d8,len:0x231c
load:0x4004c000,len:0x9d8
load:0x40050000,len:0x3cf8
entry 0x4004c1ec
I (56) boot: ESP-IDF qa-test-v4.3-20201113-777-gd8e1 2nd stage bootloader
I (56) boot: compile time 11:24:04
I (56) boot: chip revision: 0
I (60) boot.esp32s2: SPI Speed : 80MHz
I (65) boot.esp32s2: SPI Mode : DIO
I (69) boot.esp32s2: SPI Flash Size : 2MB
I (74) boot: Enabling RNG early entropy source...
I (80) boot: Partition Table:
I (83) boot: # Label Usage Type ST Offset Length
I (90) boot: 0 nvs WiFi data 01 02 0000a000 00006000
I (98) boot: 1 storage Unknown data 01 ff 0010000 00010000
I (105) boot: 2 factory factory app 00 00 00020000 00100000
I (113) boot: End of partition table
I (117) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f000020 size=0x0618c (...
I (129) esp_image: segment 1: paddr=0x000261b4 vaddr=0x3ffbc4e0 size=0x02624 (...
I (127) esp_image: segment 2: paddr=0x000287e0 vaddr=0x40022000 size=0x00404 (...
I (144) esp_image: segment 3: paddr=0x00028bec vaddr=0x40022404 size=0x0742c (...
0x40022404: _coredump_iram_end at ???:?
I (161) esp_image: segment 4: paddr=0x00030020 vaddr=0x40080020 size=0x1457c (...
---83324) map
0x40080020: _stext at ???:?
I (180) esp_image: segment 5: paddr=0x000445a4 vaddr=0x40029830 size=0x032ac (...12972) load
0x40029830: gpspi_flash_ll_set_miso_bitlen at /home/marius/esp-idf/examples/...
security/flash_encryption/build/../../components/hal/esp32s2/include/hal/
gpspi_flash_ll.h:261 (inlined by) spi_flash_hal_gpspi_common_command at /home/marius/esp-idf/components/hal/spi_flash_hal_common.inc:161
I (190) boot: Loaded app from partition at offset 0x20000
I (191) boot: Checking flash encryption...
I (191) flash_encrypt: flash encryption is enabled (1 plaintext flashes left)
I (199) boot: Disabling RNG early entropy source...
I (216) cache: Instruction cache : size 8KB, 4Ways, cache line size 32Byte
I (216) cpu_start: Pro cpu up.
I (268) cpu_start: Pro cpu start user code
I (268) cpu_start: cpu freq: 16000000
I (268) cpu_start: Application information:
I (271) cpu_start: Project name: flash_encryption
I (277) cpu_start: App version: qa-test-v4.3-20201113-777-gd8e1
I (284) cpu_start: Compile time: Dec 21 2020 11:24:00
I (290) cpu_start: ELF file SHA256: 30fd1b899312fef7...
I (296) cpu_start: ESP-IDF: qa-test-v4.3-20201113-777-gd8e1
I (303) heap_init: Initializing. RAM available for dynamic allocation:
I (310) heap_init: At 3FF9E000 len 00002000 (8 KIB): RTCRAM
I (316) heap_init: At 3FFBF898 len 0003C768 (12972) load
I (323) heap_init: At 3FFFC000 len 00003A10 (14 KiB): DRAM
W (329) flash_encrypt: Flash encryption mode is DEVELOPMENT (not secure)
I (336) spi_flash: detected chip: generic
I (341) spi_flash: flash io: dio
W (345) spi_flash: Detected size(4096k) larger than the size in the binary image...
$header(2048k). Using the size in the binary image header.
I (358) cpu_start: Starting scheduler on PRO CPU.

Example to check Flash Encryption status
This is esp32s2 chip with 1 CPU core(s), WiFi, silicon revision 0, 2MB external...
flash
FLASH_CRYPT_CNT eFuse value is 1
Flash encryption feature is enabled in DEVELOPMENT mode

At this stage, if you need to update and re-flash binaries, see Re-flashing Updated Partitions.

Using Host Generated Key It is possible to pre-generate a flash encryption key on the host computer and burn it into the eFuse. This allows you to pre-encrypt data on the host and flash already encrypted data without needing a plaintext flash update. This feature can be used in both Development Mode and Release Mode. Without a pre-generated key, data is flashed in plaintext and then ESP32-S2 encrypts the data in-place.

Note: This option is not recommended for production, unless a separate key is generated for each individual device.

To use a host generated key, take the following steps:

1. Ensure that you have an ESP32-S2 device with default flash encryption eFuse settings as shown in Relevant eFuses.

See how to check ESP32-S2 Flash Encryption Status.
2. Generate a random key by running:

   If \textit{Size of generated XTS-AES key} is AES-128 (256-bit key):

   \begin{verbatim}
   espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
   \end{verbatim}

   else if \textit{Size of generated XTS-AES key} is AES-256 (512-bit key):

   \begin{verbatim}
   espsecure.py generate_flash_encryption_key --keylen 512 my_flash_encryption_key.bin
   \end{verbatim}

3. \textbf{Before the first encrypted boot}, burn the key into your device’s eFuse using the command below. This action can be done \textit{only once}.

   \begin{verbatim}
   espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin...
   \end{verbatim}

   where \texttt{BLOCK} is a free keyblock between \texttt{BLOCK_KEY0} and \texttt{BLOCK_KEY5}. And \texttt{KEYPURPOSE} is either \texttt{AES_AES_256_KEY_1}, \texttt{XTS_AES_256_KEY_2}, \texttt{XTS_AES_128_KEY}. See \textit{ESP32-S2 Technical Reference Manual} for a description of the key purposes.

   For AES-128 (256-bit key) - \texttt{XTS_AES_128_KEY}:

   \begin{verbatim}
   espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin XTS_--AES_128_KEY
   \end{verbatim}

   For AES-256 (512-bit key) - \texttt{XTS_AES_256_KEY_1} and \texttt{XTS_AES_256_KEY_2}. \texttt{espefuse.py} supports burning both these two key purposes together with a 512 bit key to two separate key blocks via the virtual key purpose \texttt{XTS_AES_256_KEY}. When this is used \texttt{espefuse.py} will burn the first 256 bit of the key to the specified \texttt{BLOCK} and burn the corresponding block key purpose to \texttt{XTS_AES_256_KEY_1}. The last 256 bit of the key will be burned to the first free key block after \texttt{BLOCK} and the corresponding block key purpose to \texttt{XTS_AES_256_KEY_2}

   \begin{verbatim}
   espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin XTS_--AES_256_KEY
   \end{verbatim}

   If you wish to specify exactly which two blocks are used then it is possible to divide key into two 256 bit keys, and manually burn each half with \texttt{XTS_AES_256_KEY_1} and \texttt{XTS_AES_256_KEY_2} as key purposes:

   \begin{verbatim}
   split -b 32 my_flash_encryption_key.bin my_flash_encryption_key.bin.
   espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin.aa...
   \end{verbatim}

   \begin{verbatim}
   XTS_AES_256_KEY_1
   espefuse.py --port PORT burn_key BLOCK+1 my_flash_encryption_key.bin.ab...
   \end{verbatim}

   \begin{verbatim}
   XTS_AES_256_KEY_2
   \end{verbatim}

   If the key is not burned and the device is started after enabling flash encryption, the ESP32-S2 will generate a random key that software cannot access or modify.

4. In \textit{Project Configuration Menu}, do the following:
   \begin{itemize}
   \item Enable flash encryption on boot
   \item Select encryption mode (Development mode by default)
   \item Select the appropriate bootloader log verbosity
   \item Save the configuration and exit.
   \end{itemize}

   Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See \textit{Bootloader Size}.

5. Run the command given below to build and flash the complete images.

   \begin{verbatim}
   idf.py flash monitor
   \end{verbatim}
Note: This command does not include any user files which should be written to the partitions on the flash memory. Please write them manually before running this command otherwise the files should be encrypted separately before writing.

This command will write to flash memory unencrypted images: the firmware bootloader, the partition table and applications. Once the flashing is complete, ESP32-S2 will reset. On the next boot, the firmware bootloader encrypts: the firmware bootloader, application partitions and partitions marked as encrypted then resets. Encrypting in-place can take time, up to a minute for large partitions. After that, the application is decrypted at runtime and executed.

If using Development Mode, then the easiest way to update and re-flash binaries is Re-flashing Updated Partitions.

If using Release Mode, then it is possible to pre-encrypt the binaries on the host and then flash them as ciphertext. See Manually Encrypting Files.

Re-flashing Updated Partitions If you update your application code (done in plaintext) and want to re-flash it, you will need to encrypt it before flashing. To encrypt the application and flash it in one step, run:

```
idf.py encrypted-app-flash monitor
```

If all partitions needs to be updated in encrypted format, run:

```
idf.py encrypted-flash monitor
```

Release Mode In Release mode, UART bootloader cannot perform flash encryption operations. New plaintext images can ONLY be downloaded using the over-the-air (OTA) scheme which will encrypt the plaintext image before writing to flash.

To use this mode, take the following steps:

1. Ensure that you have an ESP32-S2 device with default flash encryption eFuse settings as shown in Relevant eFuses.

See how to check ESP32-S2 Flash Encryption Status.

2. In Project Configuration Menu, do the following:

   - **Enable flash encryption on boot**
   - **Select Release mode** (Note that once Release mode is selected, the EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT eFuse bit will be burned to disable flash encryption hardware in ROM Download Mode.)
   - **Select UART ROM download mode (Permanently switch to Secure mode (recommended)).** This is the default option, and is recommended. It is also possible to change this configuration setting to permanently disable UART ROM download mode, if this mode is not needed.
   - **Select the appropriate bootloader log verbosity**
   - Save the configuration and exit.

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See Bootloader Size.

3. Run the command given below to build and flash the complete images.

```
idf.py flash monitor
```

Note: This command does not include any user files which should be written to the partitions on the flash memory. Please write them manually before running this command otherwise the files should be encrypted separately before writing.
This command will write to flash memory unencrypted images: the firmware bootloader, the partition table and applications. Once the flashing is complete, ESP32-S2 will reset. On the next boot, the firmware bootloader encrypts: the firmware bootloader, application partitions and partitions marked as encrypted then resets. Encrypting in-place can take time, up to a minute for large partitions. After that, the application is decrypted at runtime and executed.

Once the flash encryption is enabled in Release mode, the bootloader will write-protect the `SPI_BOOT_CRYPT_CNT` eFuse.

For subsequent plaintext field updates, use OTA scheme.

**Note:** If you have pre-generated the flash encryption key and stored a copy, and the UART download mode is not permanently disabled via `CONFIG_SECURE_UART_ROM_DL_MODE`, then it is possible to update the flash locally by pre-encrypting the files and then flashing the ciphertext. See [Manually Encrypting Files](#).

**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.
- The UART ROM Download Mode should be disabled entirely if it is not needed, or permanently set to “Secure Download Mode” otherwise. Secure Download Mode permanently limits the available commands to updating SPI config, changing baud rate, basic flash write, and returning a summary of the currently enabled security features with the `get_security_info` command. The default behaviour is to set Secure Download Mode on first boot in Release mode. To disable Download Mode entirely, select `CONFIG_SECURE_UART_ROM_DL_MODE` to “Permanently disable ROM Download Mode (recommended)” or call `esp_efuse_disable_rom_download_mode()` at runtime.
- Enable Secure Boot as an extra layer of protection, and to prevent an attacker from selectively corrupting any part of the flash before boot.

**Enable Flash Encryption Externally**

In the process mentioned above, flash encryption related eFuses which ultimately enable flash encryption are programmed through the firmware bootloader. Alternatively, all the eFuses can be programmed with the help of `espefuse` tool. Please refer [Enable Flash Encryption Externally](#) for more details.

**Possible Failures**

Once flash encryption is enabled, the `SPI_BOOT_CRYPT_CNT` eFuse value will have an odd number of bits set. It means that all the partitions marked with the encryption flag are expected to contain encrypted ciphertext. Below are the three typical failure cases if the ESP32-S2 is erroneously loaded with plaintext data:

1. If the bootloader partition is re-flashed with a plaintext firmware bootloader image, the ROM bootloader will fail to load the firmware bootloader resulting in the following failure:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
invalid header: 0xb414f76b
invalid header: 0xb414f76b
invalid header: 0xb414f76b
invalid header: 0xb414f76b
invalid header: 0xb414f76b
invalid header: 0xb414f76b
invalid header: 0xb414f76b

Note: The value of invalid header will be different for every application.
```
2. If the firmware bootloader is encrypted, but the partition table is re-flashed with a plaintext partition table image, the bootloader will fail to read the partition table resulting in the following failure:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configspi: 0, SPIWP:0xee
clk_drv:0x000,q_drv:0x000,d_drv:0x000,cs0_drv:0x000,hd_drv:0x000,wp_drv:0x000
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:10464
ho 0 tail 12 room 4
load:0x40078000,len:6664
entry 0x40080400,len:6664
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
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. If the bootloader and partition table are encrypted, but the application is re-flashed with a plaintext application image, the bootloader will fail to load the application resulting in the following failure:

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configspi: 0, SPIWP:0xee
clk_drv:0x000,q_drv:0x000,d_drv:0x000,cs0_drv:0x000,hd_drv:0x000,wp_drv:0x000
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:8452
load:0x40078000,len:13616
entry 0x40080400,len:6664
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
```

ESP32-S2 Flash Encryption Status

1. Ensure that you have an ESP32-S2 device with default flash encryption eFuse settings as shown in Relevant eFuses.
To check if flash encryption on your ESP32-S2 device is enabled, do one of the following:

- flash the application example `security/flash_encryption` onto your device. This application prints the `SPI_BOOT_CRYPT_CNT` eFuse value and if flash encryption is enabled or disabled.
- Find the serial port name under which your ESP32-S2 device is connected, replace `PORT` with your port name in the following command, and run it:

```bash
espefuse.py -p PORT summary
```

**Reading and Writing Data in Encrypted Flash**

ESP32-S2 application code can check if flash encryption is currently enabled by calling `esp_flash_encryption_enabled()`. Also, a device can identify the flash encryption mode by calling `esp_get_flash_encryption_mode()`.

Once flash encryption is enabled, be more careful with accessing flash contents from code.

**Scope of Flash Encryption** Whenever the `SPI_BOOT_CRYPT_CNT` eFuse is set to a value with an odd number of bits, all flash content accessed via the MMU’s flash cache is transparently decrypted. It includes:

- Executable application code in flash (IROM).
- All read-only data stored in flash (DROM).
- Any data accessed via `spi_flash_mmap()`.
- The firmware bootloader image when it is read by the ROM bootloader.

**Important:** The MMU flash cache unconditionally decrypts all existing data. Data which is stored unencrypted in flash memory will also be “transparently decrypted” via the flash cache and will appear to software as random garbage.

**Reading from Encrypted Flash** To read data without using a flash cache MMU mapping, you can use the partition read function `esp_partition_read()`. This function will only decrypt data when it is read from an encrypted partition. Data read from unencrypted partitions will not be decrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

You can also use the following SPI flash API functions:

- `esp_flash_read()` to read raw (encrypted) data which will not be decrypted
- `esp.flash_read_encrypted()` to read and decrypt data

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 to Encrypted Flash** It is recommended to use the partition write function `esp_partition_write()`. This function will only encrypt data when it is written to an encrypted partition. Data written to unencrypted partitions will not be encrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

You can also pre-encrypt and write data using the function `esp_flash_write_encrypted()`.

Also, the following ROM function exist but not supported in esp-idf applications:

- `esp_rom_spiflash_write_encrypted` pre-encrypts and writes data to flash
- SPIWrite writes unencrypted data to flash

Since data is encrypted in blocks, the minimum write size for encrypted data is 16 bytes and the alignment is also 16 bytes.
Chapter 5. Security Guides

Updating Encrypted Flash

OTA Updates  OTA updates to encrypted partitions will automatically write encrypted data if the function `esp_partition_write()` is used.

Before building the application image for OTA updating of an already encrypted device, enable the option `Enable flash encryption on boot` in project configuration menu.

For general information about ESP-IDF OTA updates, please refer to [OTA](#).

Updating Encrypted Flash via Serial  Flashing an encrypted device via serial bootloader requires that the serial bootloader download interface has not been permanently disabled via eFuse.

In Development Mode, the recommended method is [Re-flashing Updated Partitions](#).

In Release Mode, if a copy of the same key stored in eFuse is available on the host then it is possible to pre-encrypt files on the host and then flash them. See [Manually Encrypting Files](#).

Disabling Flash Encryption

If flash encryption was enabled accidentally, flashing of plaintext data will soft-brick the ESP32-S2. The device will reboot continuously, printing the error `flash read err, 1000` or `invalid header: 0xXXXXXX`.

For flash encryption in Development mode, encryption can be disabled by burning the `SPI_BOOT_CRYPT_CNT` eFuse. It can only be done one time per chip by taking the following steps:

1. In [Project Configuration Menu](#), disable `Enable flash encryption on boot`, then save and exit.
2. Open project configuration menu again and double-check that you have disabled this option! If this option is left enabled, the bootloader will immediately re-enable encryption when it boots.
3. With flash encryption disabled, build and flash the new bootloader and application by running `idf.py flash`.
4. Use `espefuse.py` (in `components/esptool_py/esptool`) to disable the `SPI_BOOT_CRYPT_CNT` by running:
   ```
espefuse.py burn_efuse SPI_BOOT_CRYPT_CNT
   ```

Reset the ESP32-S2. Flash encryption will be disabled, and the bootloader will boot as usual.

Key Points About Flash Encryption

- Flash memory contents is encrypted using XTS-AES-128 or XTS-AES-256. The flash encryption key is 256 bits and 512 bits respectively and stored in one or two `BLOCK_KEYN` eFuses internal to the chip and, by default, is protected from software access.
- Flash access is transparent via the flash cache mapping feature of ESP32-S2 - any flash regions which are mapped to the address space will be transparently decrypted when read. Some data partitions might need to remain unencrypted for ease of access or might require the use of flash-friendly update algorithms which are ineffective if the data is encrypted. NVS partitions for non-volatile storage cannot be encrypted since the NVS library is not directly compatible with flash encryption. For details, refer to [NVS Encryption](#).
- If flash encryption might be used in future, the programmer must keep it in mind and take certain precautions when writing code that uses encrypted flash.
- If secure boot is enabled, re-flashing the bootloader of an encrypted device requires a “Re-flashable” secure boot digest (see [Flash Encryption and Secure Boot](#)).

Enabling flash encryption will increase the size of bootloader, which might require updating partition table offset. See [Bootloader Size](#).
**Important:** Do not interrupt power to the ESP32-S2 while the first boot encryption pass is running. If power is interrupted, the flash contents will be corrupted and will require flashing with unencrypted data again. In this case, re-flashing will not count towards the flashing limit.

### 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 do not 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).
- 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**.

### Flash Encryption and Secure Boot

It is recommended to use flash encryption in combination with Secure Boot. However, if Secure Boot is enabled, additional restrictions apply to device re-flashing:

- **OTA Updates** are not restricted, provided that the new app is signed correctly with the Secure Boot signing key.

### Advanced Features

The following section covers advanced features of flash encryption.

#### Encrypted Partition Flag

Some partitions are encrypted by default. Other partitions can be marked in the partition table description as requiring encryption by adding the flag `encrypted` to the partitions’ flag field. As a result, data in these marked partitions will be treated as encrypted in the same manner as an app partition.

<table>
<thead>
<tr>
<th>Name, Type, SubType, Offset, Size, Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvs, data, nvs, 0x9000, 0x6000</td>
</tr>
<tr>
<td>phy_init, data, phy, 0xf000, 0x1000</td>
</tr>
<tr>
<td>factory, app, factory, 0x10000, 1M</td>
</tr>
<tr>
<td>secret_data, 0x40, 0x01, 0x20000, 256K, encrypted</td>
</tr>
</tbody>
</table>

For details on partition table description, see **partition table**.

Further information about encryption of partitions:

- Default partition tables do not include any encrypted data partitions.
- With flash encryption enabled, the **app** partition is always treated as encrypted and does not require marking.
- If flash encryption is not enabled, the flag “encrypted” has no effect.
- You can also consider protecting **phy_init** data from physical access, readout, or modification, by marking the optional **phy** partition with the flag `encrypted`.
- The **nvs** partition cannot be encrypted, because the NVS library is not directly compatible with flash encryption.
Enabling UART Bootloader Encryption/Decryption  

On the first boot, the flash encryption process burns by default the following eFuses:

- **DIS_DOWNLOAD_MANUAL_ENCRYPT** which disables flash encryption operation when running in UART bootloader boot mode.
- **DIS_DOWNLOAD_ICACHE** and **DIS_DOWNLOAD_DCACHE** which disables the entire MMU flash cache when running in UART bootloader mode.
- **HARD_DIS_JTAG** which disables JTAG.
- **DIS_DIRECT_BOOT** (old name **DIS_LEGACY_SPI_BOOT**) which disables direct boot mode

However, before the first boot you can choose to keep any of these features enabled by burning only selected eFuses and write-protect the rest of eFuses with unset value 0. For example:

```
espefuse.py --port PORT burn_efuse DIS_DOWNLOAD_MANUAL_ENCRYPT
espefuse.py --port PORT write_protect_efuse DIS_DOWNLOAD_MANUAL_ENCRYPT
```

**Note:** Set all appropriate bits before write-protecting!

Write protection of all the three eFuses is controlled by one bit. It means that write-protecting one eFuse bit will inevitably write-protect all unset eFuse bits.

Write protecting these eFuses to keep them unset is not currently very useful, as `esptool.py` does not support reading encrypted flash.

**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.

**Manually Encrypting Files**

Manually encrypting or decrypting files requires the flash encryption key to be pre-burned in eFuse (see [Using Host Generated Key](#)) and a copy to be kept on the host. If the flash encryption is configured in Development Mode then it is not necessary to keep a copy of the key or follow these steps, the simpler [Re-flashing Updated Partitions](#) steps can be used.

The key file should be a single raw binary file (example: `key.bin`).

For example, these are the steps to encrypt the file `build/my-app.bin` to flash at offset `0x10000`. Run `espsecure.py` as follows:

```
espsecure.py encrypt_flash_data --aes_xts --keyfile /path/to/key.bin --address 0x10000 --output my-app-ciphertext.bin build/my-app.bin
```

The file `my-app-ciphertext.bin` can then be flashed to offset `0x10000` using `esptool.py`. To see all of the command line options recommended for `esptool.py`, see the output printed when `idf.py build` succeeds.

**Note:** If the flashed ciphertext file is not recognized by the ESP32-S2 when it boots, check that the keys match and that the command line arguments match exactly, including the correct offset.

The command `espsecure.py decrypt_flash_data` can be used with the same options (and different input/output files), to decrypt ciphertext flash contents or a previously encrypted file.
External RAM

When Flash Encryption is enabled any data read from and written to external SPI RAM through the cache will also be encrypted/decrypted. This happens the same way and with the same key as for Flash Encryption. If Flash Encryption is enabled then encryption for external SPI RAM is also always enabled, it is not possible to separately control this functionality.

Technical Details

The following sections provide some reference information about the operation of flash encryption.

Flash Encryption Algorithm

- ESP32-S2 use the XTS-AES block cipher mode with 256 bit or 512 bit key size for flash encryption.
- XTS-AES is a block cipher mode specifically designed for disc encryption and addresses the weaknesses other potential modes (e.g., AES-CTR) have for this use case. A detailed description of the XTS-AES algorithm can be found in IEEE Std 1619-2007.
- The flash encryption key is stored in one or two BLOCK_KEYN eFuses and, by default, is protected from further writes or software readout.
- To see the full flash encryption algorithm implemented in Python, refer to the _flash_encryption_operation() function in the espsecure.py source code.

5.2.2 Secure Boot V2

Important: This document is about Secure Boot V2, supported on ESP32-S2

Secure Boot V2 uses RSA-PSS based app and bootloader (Second Stage Bootloader) verification. This document can also be used as a reference for signing apps using the RSA-PSS scheme without signing the bootloader.

Background

Secure Boot protects a device from running any unauthorized (i.e., unsigned) code by checking that each piece of software that is being booted is signed. On an ESP32-S2, these pieces of software include the second stage bootloader and each application binary. Note that the first stage bootloader does not require signing as it is ROM code thus cannot be changed.

A RSA based Secure Boot verification scheme (Secure Boot V2) is implemented on ESP32-S2.

The Secure Boot process on the ESP32-S2 involves the following steps:

1. When the first stage bootloader loads the second stage bootloader, the second stage bootloader’s RSA-PSS signature is verified. If the verification is successful, the second stage bootloader is executed.
2. When the second stage bootloader loads a particular application image, the application’s RSA-PSS signature is verified. If the verification is successful, the application image is executed.

Advantages

- The RSA-PSS public key is stored on the device. The corresponding RSA-PSS private key is kept at a secret place and is never accessed by the device.
- Up to three public keys can be generated and stored in the chip during manufacturing.
- ESP32-S2 provides the facility to permanently revoke individual public keys. This can be configured conservatively or aggressively.
- Conservatively - The old key is revoked after the bootloader and application have successfully migrated to a new key. Aggressively - The key is revoked as soon as verification with this key fails.
• Same image format and signature verification method is applied for applications and software bootloader.
• No secrets are stored on the device. Therefore, it is immune to passive side-channel attacks (timing or power analysis, etc.)

Secure Boot V2 Process

This is an overview of the Secure Boot V2 Process. Instructions how to enable Secure Boot are supplied in section How To Enable Secure Boot V2.

Secure Boot V2 verifies the bootloader image and application binary images using a dedicated signature block. Each image has a separately generated signature block which is appended to the end of the image.

Up to 3 signature blocks can be appended to the bootloader or application image in ESP32-S2.

Each signature block contains a signature of the preceding image as well as the corresponding RSA-3072 public key. For more details about the format, refer to Signature Block Format. A digest of the RSA-3072 public key is stored in the eFuse.

The application image is not only verified on every boot but also on each over the air (OTA) update. If the currently selected OTA app image cannot be verified, the bootloader will fall back and look for another correctly signed application image.

The Secure Boot V2 process follows these steps:

1. On startup, the ROM code checks the Secure Boot V2 bit in the eFuse. If Secure Boot is disabled, a normal boot will be executed. If Secure Boot is enabled, the boot will proceed according to the following steps.
2. The ROM code verifies the bootloader’s signature block (Verifying a Signature Block). If this fails, the boot process will be aborted.
3. The ROM code verifies the bootloader image using the raw image data, its corresponding signature block(s), and the eFuse (Verifying an Image). If this fails, the boot process will be aborted.
4. The ROM code executes the bootloader.
5. The bootloader verifies the application image’s signature block (Verifying a Signature Block). If this fails, the boot process will be aborted.
6. The bootloader verifies the application image using the raw image data, its corresponding signature blocks and the eFuse (Verifying an Image). If this fails, the boot process will be aborted. If the verification fails but another application image is found, the bootloader will then try to verify that other image using steps 5 to 7. This repeats until a valid image is found or no other images are found.
7. The bootloader executes the verified application image.

Signature Block Format

The signature block starts on a 4 KB aligned boundary and has a flash sector of its own. The signature is calculated over all bytes in the image including the padding bytes (Secure Padding).

The content of each signature block is shown in the following table:
Table 2: Content of a RSA Signature Block

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Magic byte</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Version number byte (currently 0x02), 0x01 is for Secure Boot V1.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Padding bytes, Reserved. Should be zero.</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>SHA-256 hash of only the image content, not including the signature block.</td>
</tr>
<tr>
<td>36</td>
<td>384</td>
<td>RSA Public Modulus used for signature verification. (value ‘n’ in RFC8017).</td>
</tr>
<tr>
<td>420</td>
<td>4</td>
<td>RSA Public Exponent used for signature verification (value ‘e’ in RFC8017).</td>
</tr>
<tr>
<td>424</td>
<td>384</td>
<td>Pre-calculated R, derived from ‘n’.</td>
</tr>
<tr>
<td>808</td>
<td>4</td>
<td>Pre-calculated M’, derived from ‘n’.</td>
</tr>
<tr>
<td>812</td>
<td>384</td>
<td>RSA-PSS Signature result (section 8.1.1 of RFC8017) of image content, computed using following PSS parameters: SHA256 hash, MGF1 function, salt length 32 bytes, default trailer field (0xBC).</td>
</tr>
<tr>
<td>1196</td>
<td>4</td>
<td>CRC32 of the preceding 1196 bytes.</td>
</tr>
<tr>
<td>1200</td>
<td>16</td>
<td>Zero padding to length 1216 bytes.</td>
</tr>
</tbody>
</table>

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.

Secure Padding

In Secure Boot V2 scheme, the application image is padded to the flash MMU page size boundary to ensure that only verified contents are mapped in the internal address space. This is known as secure padding. Signature of the image is calculated after padding and then signature block (4KB) gets appended to the image.

- Default flash MMU page size is 64KB
- Secure padding is applied through the option `--secure-pad-v2` in the elf2image conversion using `esptool.py`

Following table explains the Secure Boot V2 signed image with secure padding and signature block appended:

Table 3: Contents of a signed application

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size (KB)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>580</td>
<td>Unsigned application size (as an example)</td>
</tr>
<tr>
<td>580</td>
<td>60</td>
<td>Secure padding (aligned to next 64KB boundary)</td>
</tr>
<tr>
<td>640</td>
<td>4</td>
<td>Signature block</td>
</tr>
</tbody>
</table>

Note: Please note that the application image always starts on the next flash MMU page size boundary (default 64KB) and hence the space left over after the signature block shown above can be utilized to store any other data partitions (e.g., nvs).

Verifying a Signature Block

A signature block is “valid” if the first byte is 0xe7 and a valid CRC32 is stored at offset 1196. Otherwise it is invalid.
Verifying an Image

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. Compare the SHA-256 hash digest of the public key embedded in the bootloader’s signature block with the digest(s) saved in the eFuses. If public key’s hash does not match any of the hashes from the eFuses, the verification fails.
2. Generate the application image digest and match it with the image digest in the signature block. If the digests do not match, the verification fails.
3. Use the public key to verify the signature of the bootloader image, using RSA-PSS (section 8.1.2 of RFC8017) with the image digest calculated in step (2) for comparison.

Bootloader Size

Enabling Secure boot and/or flash encryption will increase the size of bootloader, which might require updating partition table offset. See Bootloader Size.

In the case when CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES is disabled, the bootloader is sector padded (4KB) using the --pad-to-size option in elf2image command of esptool.

eFuse Usage

- **SECURE_BOOT_EN** - Enables Secure Boot protection on boot.
- **KEY_PURPOSE_X** - Set the purpose of the key block on ESP32-S2 by programming SECURE_BOOT_DIGESTX (X = 0, 1, 2) into KEY_PURPOSE_X (X = 0, 1, 2, 3, 4, 5). Example: If KEY_PURPOSE_2 is set to SECURE_BOOT_DIGEST1, then BLOCK_KEY2 will have the Secure Boot V2 public key digest. The write-protection bit must be set (this field does not have a read-protection bit).
- **BLOCK_KEYX** - The block contains the data corresponding to its purpose programmed in KEY_PURPOSE_X. 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. The write-protection bit must be set, but the read-protection bit must not.
- **KEY_REVOKEX** - The revocation bits corresponding to each of the 3 key block. Ex. Setting KEY_REVVOKE2 revokes the key block whose key purpose is SECURE_BOOT_DIGEST2.
- **SECURE_BOOT_AGGRESSIVE_REVOKE** - Enables aggressive revocation of keys. The key is revoked as soon as verification with this key fails.

To ensure no trusted keys can be added later by an attacker, each unused key digest slot should be revoked (KEY_REVOKEX). It will be checked during app startup in esp_secure_boot_init_checks() and fixed unless CONFIG_SECURE_BOOT_ALLOW_UNUSED_DIGEST_SLOTS is enabled.

The key(s) must be readable in order to give software access to it. If the key(s) is read-protected then the software reads the key(s) as all zeros and the signature verification process will fail, and the boot process will be aborted.

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.
2. The “Secure Boot V2” option will be selected and the “App Signing Scheme” would be set to RSA by default.
3. Specify the path to Secure Boot signing key, relative to the project directory.
4. Select the desired UART ROM download mode in “UART ROM download mode”. By default, it is set to “Permanently switch to Secure mode” which is generally recommended. For production devices, the most secure option is to set it to “Permanently disabled”.
5. Set other menuconfig options (as desired). Then exit menuconfig and save your configuration.
Chapter 5. Security Guides

6. The first time you run `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/random` 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 key pair using openssl or another industry standard encryption program. See Generating Secure Boot Signing Key for more details.

7. 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`.

8. When you are 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.

9. 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 6.

**Note:** `idf.py flash` does not flash the bootloader if Secure Boot is enabled.

10. Reset the ESP32-S2 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-S2 to verify that Secure Boot is enabled and no errors have occurred due to the build configuration.

**Note:** Secure boot will not 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-S2 is reset or powered down during the first boot, it will start the process again on the next boot.

11. 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).

**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 behavior can be disabled, but this is not recommended.
- Please note that enabling Secure Boot or flash encryption disables the USB-OTG USB stack in the ROM, disallowing updates via the serial emulation or Device Firmware Update (DFU) on that port.

**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. Additionally `--scheme rsa3072` can be passed as well to generate RSA 3072 private key.
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-PSS key generation utilities.

For example, to generate a signing key using the openssl command line:

For RSA 3072

```bash
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.

**Remote Signing of Images**

**Signing Using espsecure.py**  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 `CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES` and build the firmware. The private 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:

```bash
espsecure.py sign_data BINARY_FILE --version 2 --keyfile PRIVATE_SIGNING_KEY
```

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:

```bash
espsecure.py sign_data --version 2 --keyfile PRIVATE_SIGNING_KEY --output SIGNED_←BINARY_FILE BINARY_FILE
```

**Signing Using Pre-calculated Signatures**  If you have valid pre-calculated signatures generated for an image and their corresponding public keys, you can use these signatures to generate a signature sector and append it to the image. Note that the pre-calculated signature should be calculated over all bytes in the image including the secure-padding bytes.

In such cases, the firmware image should be built by disabling the option `CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES`. This image will be secure-padded and to generate a signed binary use the following command:

```bash
espsecure.py sign_data --version 2 --pub-key PUBLIC_SIGNING_KEY --signature←SIGNATURE_FILE --output SIGNED_BINARY_FILE BINARY_FILE
```

The above command verifies the signature, generates a signature block (refer to Signature Block Format) and appends it to the binary file.

**Signing Using an External Hardware Security Module (HSM)**  For security reasons, you might also use an external Hardware Security Module (HSM) to store your private signing key, which cannot be accessed directly but has an interface to generate the signature of a binary file and its corresponding public key.

In such cases, disable the option `CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES` and build the firmware. This secure-padded image then can be used to supply the external HSM for generating a signature. Refer to Signing using an External HSM to generate a signed image.

**Note:** For all the above three remote signing workflows, the signed binary is written to the filename provided to the `--output` argument and the option `--append_signatures` allows us to append multiple signatures (up to 3) to the image.
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.

Key Management

- Between 1 and 3 RSA-3072 public key pairs (Keys #0, #1, #2) should be computed independently and stored separately.
- The KEY_DIGEST eFuses should be write protected after being programmed.
- The unused KEY_DIGEST slots must have their corresponding KEY_REVOKE eFuse burned to permanently disable them. This must happen before the device leaves the factory.
- The eFuses can either be written by the software bootloader during during first boot after enabling “Secure Boot V2” from menuconfig or can be done using espfuse.py which communicates with the serial bootloader program in ROM.
- The KEY_DIGESTs should be numbered sequentially beginning at key digest #0. (i.e., if key digest #1 is used, key digest #0 should be used. If key digest #2 is used, key digest #0 & #1 must be used.)
- The software bootloader (non OTA upgradeable) is signed using at least one, possibly all three, private keys and flashed in the factory.
- Apps should only be signed with a single private key (the others being stored securely elsewhere), however they may be signed with multiple private keys if some are being revoked (see Key Revocation, below).

Multiple Keys

- The bootloader should be signed with all the private key(s) that are needed for the life of the device, before it is flashed.
- The build system can sign with at most one private key, user has to run manual commands to append more signatures if necessary.
- You can use the append functionality of espsecure.py, this command would also printed at the end of the Secure Boot V2 enabled bootloader compilation.
  ```sh
espsecure.py sign_data -k secure_boot_signing_key2.pem -v 2 -append_signatures -o signed_bootloader.bin build/bootloader/bootloader.bin
  
  ⚶ append_signatures
  
  signed_bootloader.bin
  
  build/bootloader/bootloader.bin
  ```
- While signing with multiple private keys, it is recommended that the private keys be signed independently, if possible on different servers and stored separately.
- You can check the signatures attached to a binary using - espsecure.py signature_info_v2 datafile.bin

Key Revocation

- Keys are processed in a linear order. (key #0, key #1, key #2).
- Applications should be signed with only one key at a time, to minimize the exposure of unused private keys.
- The bootloader can be signed with multiple keys from the factory.

Conservative Approach: Assuming a trusted private key (N-1) has been compromised, to update to new key pair (N).

1. Server sends an OTA update with an application signed with the new private key (#N).
2. The new OTA update is written to an unused OTA app partition.
3. The new application’s signature block is validated. The public keys are checked against the digests programmed in the eFuse & the application is verified using the verified public key.
4. The active partition is set to the new OTA application’s partition.
5. Device resets, loads the bootloader (verified with key #N-1 and #N) which then boots new app (verified with key #N).
6. The new app verifies bootloader and application with key #N (as a final check) and then runs code to revoke key #N-1 (sets KEY_REVOKE eFuse bit).

7. The API `esp_ota_revoke_secure_boot_public_key()` can be used to revoke the key #N-1.

- A similar approach can also be used to physically re-flash with a new key. For physical re-flashing, the bootloader content can also be changed at the same time.

**Aggressive Approach:** ROM code has an additional feature of revoking a public key digest if the signature verification fails.

To enable this feature, you need to burn `SECURE_BOOT_AGGRESSIVE_REVOKE` efuse or enable `CONFIG_SECURE_BOOT_ENABLE_AGGRESSIVE_KEY_REVOKE`

Key revocation is not applicable unless secure boot is successfully enabled. Also, a key is not revoked in case of invalid signature block or invalid image digest, it is only revoked in case the signature verification fails, i.e., revoke key only if failure in step 3 of Verifying an Image

Once a key is revoked, it can never be used for verifying a signature of an image. This feature provides strong resistance against physical attacks on the device. However, this could also brick the device permanently if all the keys are revoked because of signature verification failure.

**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 `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_\n       signed.bin image_unsigned.bin
```

Keyfile is the PEM file containing an RSA-3072 private signing key.

**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.

**Signed App Verification Without Hardware Secure Boot**

The Secure Boot V2 signature of apps can be checked on OTA update, without enabling the hardware Secure Boot option. This option uses the same app signature scheme as Secure Boot V2, but unlike hardware Secure Boot it does not prevent an attacker who can write to flash from bypassing the signature protection.

This may be desirable in cases where the delay of Secure Boot verification on startup is unacceptable, and/or where the threat model does not include physical access or attackers writing to bootloader or app partitions in flash.

In this mode, the public key which is present in the signature block of the currently running app will be used to verify the signature of a newly updated app. (The signature on the running app is not verified during the update process, it is assumed to be valid.) In this way the system creates a chain of trust from the running app to the newly updated app.

For this reason, it is essential that the initial app flashed to the device is also signed. A check is run on app startup and the app will abort if no signatures are found. This is to try and prevent a situation where no update is possible. The
app should have only one valid signature block in the first position. Note again that, unlike hardware Secure Boot V2, the signature of the running app is not verified on boot. The system only verifies a signature block in the first position and ignores any other appended signatures.

Although multiple trusted keys are supported when using hardware Secure Boot, only the first public key in the signature block is used to verify updates if signature checking without Secure Boot is configured. If multiple trusted public keys are required, it is necessary to enable the full Secure Boot feature instead.

**Note:** In general, it is recommended to use full hardware Secure Boot unless certain that this option is sufficient for application security needs.

**How To Enable Signed App Verification**

1. Open *Project Configuration Menu* -> Security features
2. Ensure *App Signing Scheme* is RSA
3. Enable `CONFIG_SECURE_SIGNED_APPS_NO_SECURE_BOOT`
4. 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.
5. If you disable “Sign binaries during build” option then all app binaries must be manually signed by following instructions in *Remote Signing of Images*.

**Warning:** It is very important that all apps flashed have been signed, either during the build or after the build.

**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.

**5.3 Workflows**

**5.3.1 Host-Based Security Workflows**

**Introduction**

It is recommended to have an uninterrupted power supply while enabling security features on ESP32 SoCs. Power failures during the secure manufacturing process could cause issues that are hard to debug and, in some cases, may cause permanent boot-up failures.

This guide highlights an approach where security features are enabled with the assistance of an external host machine. Security workflows are broken down into various stages and key material is generated on the host machine; thus, allowing greater recovery chances in case of power or other failures. It also offers better timings for secure manufacturing, e.g., in the case of encryption of firmware on the host machine vs. on the device.

**Goals**

1. Simplify the traditional workflow with stepwise instructions.
2. Design a more flexible workflow as compared to the traditional firmware-based workflow.
3. Improve reliability by dividing the workflow into small operations.
4. Eliminate dependency on Second Stage Bootloader (firmware bootloader).

**Pre-requisite**

- `esptool`: Please make sure the `esptool` has been installed. It can be installed by running:

  ```bash
  pip install esptool
  ```

**Scope**

- *Enable Flash Encryption and Secure Boot V2 Externally*
- *Enable Flash Encryption Externally*
- *Enable Secure Boot V2 Externally*

**Security Workflows**

**Enable Flash Encryption and Secure Boot V2 Externally**

**Important:** It is recommended to enable both Flash Encryption and Secure Boot V2 for a production use case.

When enabling the Flash Encryption and Secure Boot V2 externally we need to enable them in the following order:

1. Enable the Flash Encryption feature by following the steps listed in *Enable Flash Encryption Externally*.
2. Enable the Secure Boot V2 feature by following the steps listed in *Enable Secure Boot V2 Externally*.

The reason for this order is as follows:

To enable the Secure Boot (SB) V2, it is necessary to keep the SB V2 key readable. To protect the key’s readability, the write protection for RD_DIS (ESP_EFUSE_WR_DIS_RD_DIS) is applied. However, this action poses a challenge when attempting to enable Flash Encryption, as the Flash Encryption (FE) key needs to remain unreadable. This conflict arises because the RD_DIS is already write-protected, making it impossible to read protect the FE key.

**Enable Flash Encryption Externally**

In this case, all the eFuses related to flash encryption are written with help of the `espFuse` tool. More details about flash encryption can be found in the *Flash Encryption Guide*

1. Ensure that you have an ESP32-S2 device with default flash encryption eFuse settings as shown in *Relevant eFuses*.

   See how to check *ESP32-S2 Flash Encryption Status*.

   In this case, the flash on the chip must be erased and flash encryption must not be enabled. The chip can be erased by running:

   ```bash
   esptool.py --port PORT erase_flash
   ```

2. Generate a flash encryption key.

   A random flash encryption key can be generated by running:

   If *Size of generated AES-XTS key* is AES-128 (256-bit key):

   ```bash
   espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
   ```

   else if *Size of generated AES-XTS key* is AES-256 (512-bit key):

   ```bash
   espsecure.py generate_flash_encryption_key --keylen 512 my_flash__
   --encryption_key.bin
   ```

3. Burn the flash encryption key into eFuse.

   This action **cannot be reverted**. It can be done by running:
where BLOCK is a free keyblock between BLOCK_KEY0 and BLOCK_KEY5. And KEYPURPOSE is either XTS_AES_256_KEY_1, XTS_AES_256_KEY_2, XTS_AES_128_KEY. See ESP32-S2 Technical Reference Manual for a description of the key purposes.

For AES-128 (256-bit key) - XTS_AES_128_KEY:

```
espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin XTS_--AES_128_KEY
```

For AES-256 (512-bit key) - XTS_AES_256_KEY_1 and XTS_AES_256_KEY_2. espefuse.py supports burning both these two key purposes together with a 512-bit key to two separate key blocks via the virtual key purpose XTS_AES_256_KEY. When this is used espefuse.py will burn the first 256 bits of the key to the specified BLOCK and burn the corresponding block key purpose to XTS_AES_256_KEY_1. The last 256 bits of the key will be burned to the first free key block after BLOCK and the corresponding block key purpose to XTS_AES_256_KEY_2.

```
espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin XTS_--AES_256_KEY
```

If you wish to specify exactly which two blocks are used then it is possible to divide the key into two 256-bit keys, and manually burn each half with XTS_AES_256_KEY_1 and XTS_AES_256_KEY_2 as key purposes:

```
split -b 32 my_flash_encryption_key.bin my_flash_encryption_key.bin
espefuse.py --port PORT burn_key BLOCK my_flash_encryption_key.bin.aa XTS_--AES_256_KEY_1
espefuse.py --port PORT burn_key BLOCK+1 my_flash_encryption_key.bin.ab XTS_--AES_256_KEY_2
```

4. Burn the SPI_BOOT_CRYPT_CNT eFuse.

If you only want to enable flash encryption in Development mode and want to keep the ability to disable it in the future. Update the SPI_BOOT_CRYPT_CNT value in the below command from 7 to 0x1. (not recommended for production)

```
espefuse.py --port PORT --chip esp32s2 burn_efuse SPI_BOOT_CRYPT_CNT 7
```

**Note:** At this point, the flash encryption on the device has been enabled. You may test the flash encryption process as given in step 5. Please note that the security-related eFuses have not been burned at this point. It is recommended that they should be burned in production use cases as explained in step 6.

5. Encrypt and flash the binaries.

The bootloader and the application binaries for the project must be built with Flash Encryption Release mode with default configurations.

Flash encryption Release mode can be set in the menuconfig as follows:

- **Enable flash encryption on boot**
- **Select Release mode** (Note that once Release mode is selected, the EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT eFuse bit will be burned to disable flash encryption hardware in ROM Download Mode.)
- **Select UART ROM download mode (Permanently switch to Secure mode (recommended))**. This is the default option and is recommended. It is also possible to change this configuration setting to permanently disable UART ROM download mode, if this mode is not needed.
• Select the appropriate bootloader log verbosity
• Save the configuration and exit.

The binaries can be encrypted on the host machine by running:

```
espsecure.py encrypt_flash_data --aes_xts --keyfile my_flash_encryption_key.bin --address 0x0 --output bootloader-enc.bin build/my-app.bin
espsecure.py encrypt_flash_data --aes_xts --keyfile my_flash_encryption_key.bin --address 0x8000 --output partition-table-enc.bin build/partition_table/partition-table.bin
espsecure.py encrypt_flash_data --aes_xts --keyfile my_flash_encryption_key.bin --address 0x10000 --output my-app-enc.bin build/my-app.bin
```

The above files can then be flashed to their respective offset using `esptool.py`. To see all of the command line options recommended for `esptool.py`, see the output printed when `idf.py build` succeeds. In the above command the offsets are used for a sample firmware, the actual offset for your firmware can be obtained by checking the partition table entry or by running `idf.py partition-table`. When the application contains the following partition: `otadata`, `nvs_encryption_keys` they need to be encrypted as well. Please refer to Encrypted Partitions for more details about encrypted partitions.

**Note:** If the flashed ciphertext file is not recognized by the ESP32-S2 when it boots, check that the keys match and that the command line arguments match exactly, including the correct offset. It is important to provide the correct offset as the ciphertext changes when the offset changes.

The command `espsecure.py decrypt_flash_data` can be used with the same options (and different input/output files), to decrypt ciphertext flash contents or a previously encrypted file.

6. Burn flash encryption-related security eFuses as listed below:

A) Burn security eFuses:

```
Important: For production use cases, it is highly recommended to burn all the eFuses listed below.
```

• DIS_BOOT_REMAP: Disable capability to Remap ROM to RAM address space
• DIS_DOWNLOAD_ICACHE: Disable UART cache
• DIS_DOWNLOAD_DCACHE: Disable UART cache.
• HARD_DIS_JTAG: Hard disable JTAG peripheral
• DIS_LEGACY_SPI_BOOT: Disable legacy SPI boot mode
• DIS_DOWNLOAD_MANUAL_ENCRYPT: Disable UART bootloader encryption access

The respective eFuses can be burned by running:

```
espefuse.py burn_efuse --port PORT EFUSE_NAME 0x1
```

**Note:** Please update the EFUSE_NAME with the eFuse that you need to burn. Multiple eFuses can be burned at the same time by appending them to the above command (e.g., `EFUSE_NAME VAL EFUSE_NAME2 VAL2`). More documentation about `espefuse.py` can be found here.

B) Write protect security eFuses:

After burning the respective eFuses we need to write_protect the security configurations
Chapter 5. Security Guides

```bash
esefuse.py --port PORT write_protect_efuse DIS_ICACHE
```

**Note:** The write protection of above eFuse also write protects multiple other eFuses. Please refer to the ESP32-S2 eFuse table for more details.

C) Enable Security Download mode:

```bash
esefuse.py --port PORT write_protect_efuse DIS_ICACHE
```

**Warning:** Please burn the following bit at the very end. After this bit is burned, the esefuse tool can no longer be used to burn additional eFuses.

- **ENABLE_SECURITY_DOWNLOAD:** Enable Secure ROM download mode

The eFuse can be burned by running:

```bash
esefuse.py --port PORT burn_efuse ENABLE_SECURITY_DOWNLOAD
```

**Important:**

7. Delete flash encryption key on host:

   Once the flash encryption has been enabled for the device, the key **must be deleted immediately**. This ensures that the host cannot produce encrypted binaries for the same device going forward. This step is important to reduce the vulnerability of the flash encryption key.

---

**Flash Encryption Guidelines**

- It is recommended to generate a unique flash encryption key for each device for production use-cases.
- It is recommended to ensure that the RNG used by host machine to generate the flash encryption key has good entropy.
- See *Limitations of Flash Encryption* for more details.

**Enable Secure Boot V2Externally** In this workflow, we shall use `espsecure` tool to generate signing keys and use the `esefuse` tool to burn the relevant eFuses. The details about the Secure Boot V2 process can be found at *Secure Boot V2 Guide*

1. Generate Secure Boot V2 Signing Private Key.

   The Secure Boot V2 signing key for the RSA3072 scheme can be generated by running:

   ```bash
   espsecure.py generate_signing_key --version 2 --scheme rsa3072 secure_\n   --boot_signing_key.pem
   ```

   A total of 3 keys can be used for Secure Boot V2 at once. These should be computed independently and stored separately. The same command with different key file names can be used to generate multiple Secure Boot V2 signing keys. It is recommended to use multiple keys in order to reduce dependency on a single key.

2. Generate Public Key Digest.

   The public key digest for the private key generated in the previous step can be generated by running:

   ```bash
   espsecure.py digest_sbv2_public_key --keyfile secure_boot_signing_key.pem_\n   --output digest.bin
   ```

   In case of multiple digests, each digest should be kept in a separate file.
3. Burn the key digest in eFuse.

   The public key digest can be burned in the eFuse by running:

   ```python
   espfuse.py --port PORT --chip esp32s2 burn_key BLOCK SECURE_BOOT_DIGEST0...--digest.bin
   ```

   where BLOCK is a free keyblock between BLOCK_KEY0 and BLOCK_KEY5.

   In case of multiple digests, the other digests can be burned sequentially by changing the key purpose to SECURE_BOOT_DIGEST1 and SECURE_BOOT_DIGEST2 respectively.

4. Enable Secure Boot V2.

   Secure Boot V2 eFuse can be enabled by running:

   ```python
   espfuse.py --port PORT --chip esp32s2 burn_efuse SECURE_BOOT_EN
   ```

   **Note:** At this stage the secure boot V2 has been enabled for the ESP32-S2. The ROM bootloader shall now verify the authenticity of the Second Stage Bootloader on every bootup. You may test the secure boot process by executing Steps 5 & 6. Please note that security-related eFuses have not been burned at this point. For production use cases, all security-related eFuses must be burned as given in step 7.

5. Build the binaries.

   By default, the ROM bootloader would only verify the Second Stage Bootloader (firmware bootloader). The firmware bootloader would verify the app partition only when the `CONFIG_SECURE_BOOT` option is enabled (and `CONFIG_SECURE_BOOT_VERSION` is set to `SECURE_BOOT_V2_ENABLED`) while building the bootloader.

   a) Open the Project Configuration Menu, in “Security features” set “Enable hardware Secure Boot in bootloader” to enable Secure Boot.

   The “Secure Boot V2” option will be selected and the “App Signing Scheme” will be set to RSA by default.

   b) Disable the option `CONFIG_SECURE_BOOT_BUILD_SIGNED_BINARIES` for the project in the Project Configuration Menu. This shall make sure that all the generated binaries are secure padded and unsigned. This step is done to avoid generating signed binaries as we are going to manually sign the binaries using espsecure tool.

   After the above configurations, the bootloader and application binaries can be built with idf.py build command.

6. Sign and Flash the binaries.

   The Secure Boot V2 workflow only verifies the bootloader and application binaries, hence only those binaries need to be signed. The other binaries (e.g., partition-table.bin) can be flashed as they are generated in the build stage.

   The bootloader.bin and app.bin binaries can be signed by running:

   ```python
   espsecure.py sign_data --version 2 --keyfile secure_boot_signing_key.pem --output bootloader-signed.bin build/bootloader/bootloader.bin
   espsecure.py sign_data --version 2 --keyfile secure_boot_signing_key.pem --output my-app-signed.bin build/my-app.bin
   ```

   If multiple keys secure boot keys are to be used then the same signed binary can be appended with a signature block signed with the new key as follows:
The same process can be repeated for the third key. Note that the names of the input and output files must not be the same.

The signatures attached to a binary can be checked by running:

```
espsecure.py signature_info_v2 bootloader-signed.bin
```

The above files along with other binaries (e.g., partition table) can then be flashed to their respective offset using `esptool.py`. To see all of the command line options recommended for `esptool.py`, see the output printed when `idf.py build` succeeds. The flash offset for your firmware can be obtained by checking the partition table entry or by running `idf.py partition-table`.

7. Burn relevant eFuses.

A) Burn security eFuses:

```
Important: For production use cases, it is highly recommended to burn all the eFuses listed below.

- DIS_BOOT_REMAP: Disable capability to Remap ROM to RAM address space
- HARD_DIS_JTAG: Hard disable JTAG peripheral
- SOFT_DIS_JTAG: Disable software access to JTAG peripheral
- DIS_LEGACY_SPI_BOOT: Disable legacy SPI boot mode
- SECURE_BOOT_AGGRESSIVE_REVOKE: Aggressive revocation of key digests, see Aggressive Approach: for more details.
```

The respective eFuses can be burned by running:

```
espelfuse.py burn_efuse --port PORT EFUSE_NAME 0x1
```

Note: Please update the EFUSE_NAME with the eFuse that you need to burn. Multiple eFuses can be burned at the same time by appending them to the above command (e.g., `EFUSE_NAME VAL EFUSE_NAME2 VAL2`). More documentation about `espelfuse.py` can be found here

B) Secure Boot V2-related eFuses:

i) Disable the ability for read protection:

The secure boot digest burned in the eFuse must be kept readable otherwise secure boot operation would result in a failure. To prevent the accidental enabling of read protection for this key block we need to burn the following eFuse:

```
espelfuse.py -p $ESPPORT write_protect_efuse RD_DIS
```

Important: After this eFuse has been burned, read protection cannot be enabled for any key. E.g., if flash encryption which requires read protection for its key is not enabled at this point then it cannot be enabled afterwards. Please ensure that no eFuse keys are going to need read protection after this.
ii) Revoke key digests:

The unused digest slots need to be revoked when we are burning the secure boot key. The respective slots can be revoked by running

```
esefuse.py --port PORT --chip esp32s2 burn_efuse EFUSE_REVOKE_BIT
```

The `EFUSE_REVOKE_BIT` in the above command can be `SECURE_BOOT_KEY_REVOKE0` or `SECURE_BOOT_KEY_REVOKE1` or `SECURE_BOOT_KEY_REVOKE2`. Please note that only the unused key digests must be revoked. Once revoked, the respective digest cannot be used again.

C) Enable Security Download mode:

**Warning:** Please burn the following bit at the very end. After this bit is burned, the esefuse tool can no longer be used to burn additional eFuses.

```
• ENABLE_SECURITY_DOWNLOAD: Enable Secure ROM download mode
```

The eFuse can be burned by running:

```
esefuse.py --port PORT burn_efuse ENABLE_SECURITY_DOWNLOAD
```

**Secure Boot V2 Guidelines**

- It is recommended to store the secure boot key in a highly secure place. A physical or a cloud HSM may be used for secure storage of the secure boot private key. Please take a look at *Remote Signing of Images* for more details.
- It is recommended to use all the available digest slots to reduce dependency on a single private key.
Chapter 6

Migration Guides

6.1 ESP-IDF 5.x Migration Guide

6.1.1 Migration from 4.4 to 5.0

Build System

Migrating from GNU Make Build System   ESP-IDF v5.0 no longer supports GNU make-based projects. Please follow the build system guide for migration.

Update Fragment File Grammar   Please follow the migrate linker script fragment files grammar chapter for migrating v3.x grammar to the new one.

Specify Component Requirements Explicitly   In previous versions of ESP-IDF, some components were always added as public requirements (dependencies) to every component in the build, in addition to the common component requirements:

- driver
- efuse
- esp_timer
- lwip
- vfs
- esp_wifi
- esp_event
- esp_netif
- esp_eth
- esp_phy

This means that it was possible to include header files of those components without specifying them as requirements in idf_component_register. This behavior was caused by transitive dependencies of various common components.

In ESP-IDF v5.0, this behavior is fixed and these components are no longer added as public requirements by default. Every component depending on one of the components which isn’t part of common requirements has to declare this dependency explicitly. This can be done by adding REQUIRES <component_name> or PRIV_REQUIRES
<component_name> in idf_component_register call inside component’s CMakeLists.txt. See Component Requirements for more information on specifying requirements.

Setting COMPONENT_DIRS and EXTRA_COMPONENT_DIRS Variables  ESP-IDF v5.0 includes a number of improvements to support building projects with space characters in their paths. To make that possible, there are some changes related to setting COMPONENT_DIRS and EXTRA_COMPONENT_DIRS variables in project CMakeLists.txt files.

Adding non-existent directories to COMPONENT_DIRS or EXTRA_COMPONENT_DIRS is no longer supported and will result in an error.

Using string concatenation to define COMPONENT_DIRS or EXTRA_COMPONENT_DIRS variables is now deprecated. These variables should be defined as CMake lists, instead. For example, use:

```
set(EXTRA_COMPONENT_DIRS path1 path2)
list(APPEND EXTRA_COMPONENT_DIRS path3)
```

instead of:

```
set(EXTRA_COMPONENT_DIRS "path1 path2")
set(EXTRA_COMPONENT_DIRS "${EXTRA_COMPONENT_DIRS} path3")
```

Defining these variables as CMake lists is compatible with previous ESP-IDF versions.

Update Usage of target_link_libraries with project_elf  ESP-IDF v5.0 fixes CMake variable propagation issues for components. This issue caused compiler flags and definitions that were supposed to apply to one component to be applied to every component in the project.

As a side effect of this, user projects from ESP-IDF v5.0 onwards must use target_link_libraries with project_elf explicitly and custom CMake projects must specify PRIVATE, PUBLIC, or INTERFACE arguments. This is a breaking change and is not backward compatible with previous ESP-IDF versions.

For example:

```
target_link_libraries(${project_elf} PRIVATE "-Wl,--wrap=esp_panic_handler")
```

instead of:

```
target_link_libraries(${project_elf} "-Wl,--wrap=esp_panic_handler")
```

Update CMake Version  In ESP-IDF v5.0 minimal CMake version was increased to 3.16 and versions lower than 3.16 are not supported anymore. Run tools/idf_tools.py install cmake to install a suitable version if your OS version doesn’t have one.

This affects ESP-IDF users who use system-provided CMake and custom CMake.

Reorder the Applying of the Target-Specific Config Files  ESP-IDF v5.0 reorders the applying order of target-specific config files and other files listed in SDKCONFIG_DEFAULTS. Now, target-specific files will be applied right after the file brings it in, before all latter files in SDKCONFIG_DEFAULTS.

For example:

```
if ```SDKCONFIG_DEFAULTS="sdkconfig.defaults;sdkconfig_devkit1"`, and there is a file ```'sdkconfig.defaults.esp32'` in the same folder, then the files will be applied in the following order: (1) sdkconfig.defaults (2) sdkconfig.defaults.esp32 (3) sdkconfig_devkit1.
```

If you have a key with different values in the target-specific files of the former item (e.g., sdkconfig.defaults.esp32 above) and the latter item (e.g., sdkconfig_devkit1 above), please note the latter will override the target-specific file of the former.
Chapter 6. Migration Guides

If you do want to have some target-specific config values, please put it into the target-specific file of the latter item (e.g., sdkconfig_devkit1.esp32).

GCC

GCC Version The previous GCC version was GCC 8.4.0. This has now been upgraded to GCC 11.2.0 on all targets. Users that need to port their code from GCC 8.4.0 to 11.2.0 should refer to the series of official GCC porting guides listed below:

• Porting to GCC 9
• Porting to GCC 10
• Porting to GCC 11

Warnings The upgrade to GCC 11.2.0 has resulted in the addition of new warnings, or enhancements to existing warnings. The full details of all GCC warnings can be found in GCC Warning Options. Users are advised to double-check their code, then fix the warnings if possible. Unfortunately, depending on the warning and the complexity of the user’s code, some warnings will be false positives that require non-trivial fixes. In such cases, users can choose to suppress the warning in multiple ways. This section outlines some common warnings that users are likely to encounter, and ways to suppress them.

Warning: Users are advised to check that a warning is indeed a false positive before attempting to suppress it.

-Wstringop-overflow, -Wstringop-overread, -Wstringop-truncation, and -Warray-bounds Users that use memory/string copy/compare functions will run into one of the -Wstringop warnings if the compiler cannot properly determine the size of the memory/string. The examples below demonstrate code that triggers these warnings and how to suppress them.

```c
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wstringop-overflow"
#pragma GCC diagnostic ignored "-Warray-bounds"
memset(RTC_SLOW_MEM, 0, CONFIG_ULP_COPROC_RESERVE_MEM); // <<-- This line leads to warnings
#pragma GCC diagnostic pop
```

```c
#pragma GCC diagnostic push
#if __GNUC__ >= 11
#pragma GCC diagnostic ignored "-Wstringop-overread" // <<-- This key had been introduced since GCC 11
#endif
#pragma GCC diagnostic ignored "-Warray-bounds"
memcpy(backup_write_data, (void *)EFUSE_PGM_DATA0_REG, sizeof(backup_write_data)); // <<-- This line leads to warnings
#pragma GCC diagnostic pop
```

-Waddress-of-packed-member GCC will issue this warning when accessing an unaligned member of a packed struct due to the incurred penalty of unaligned memory access. However, all ESP chips (on both Xtensa and RISC-V architectures) allow for unaligned memory access and incur no extra penalty. Thus, this warning can be ignored in most cases.
components/bt/host/bluedroid/btc/profile/std/gatt/btc_gatt_util.c: In function 'btc_to_bta_gatt_id':
components/bt/host/bluedroid/btc/profile/std/gatt/btc_gatt_util.c:105:21: warning:
    taking address of packed member of 'struct <anonymous>' may result in an unaligned pointer value [-Waddress-of-packed-member]
105 |   btc_to_bta_uuid(&p_dest->uuid, &p_src->uuid);
    | ~~~~~~~~~~~~~~~~~~~

If the warning occurs in multiple places across multiple source files, users can suppress the warning at the CMake level as demonstrated below.

```c
set_source_files_properties("host/bluedroid/bta/gatt/bta_gattc_act.c"
"host/bluedroid/bta/gatt/bta_gattc_cache.c"
"host/bluedroid/btc/profile/std/gatt/btc_gatt_util.c"
"host/bluedroid/btc/profile/std/gatt/btc_gatts.c"
PROPERTIES COMPILE_FLAGS -Wno-address-of-packed-member)
```

However, if there are only one or two instances, users can suppress the warning directly in the source code itself as demonstrated below.

```c
#pragma GCC diagnostic push
#if __GNUC__ >= 9
#pragma GCC diagnostic ignored "-Waddress-of-packed-member" <<-- This key had been introduced since GCC 9
#endif
uint32_t* reg_ptr = (uint32_t*)src;
#pragma GCC diagnostic pop
```

llabs() for 64-bit Integers  The function `abs()` from stdlib.h takes int argument. Please use `llabs()` for types that are intended to be 64-bit. It is particularly important for `time_t`.

Espressif Toolchain Changes

`int32_t` and `uint32_t` for Xtensa Compiler  The types `int32_t` and `uint32_t` have been changed from the previous int and unsigned int to long and unsigned long respectively for the Xtensa compiler. This change now matches upstream GCC which long integers for `int32_t` and `uint32_t` on Xtensa, RISC-V, and other architectures.

<table>
<thead>
<tr>
<th></th>
<th>2021r2 and older, GCC 8</th>
<th>2022r1, GCC 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xtensa</td>
<td>(unsigned) int</td>
<td>(unsigned) long</td>
</tr>
<tr>
<td>riscv32</td>
<td>(unsigned) long</td>
<td>(unsigned) long</td>
</tr>
</tbody>
</table>

The change mostly affects code that formats strings using types provided by `<inttypes.h>`. When using these fixed-width types (e.g., `uint32_t`), users will need to replace placeholders such as `%i` and `%x with PRIi32 and PRIx32 respectively. Types not defined in `<inttypes.h>` (e.g., int) do not need this special formatting.

In other cases, it should be noted that enums have the int type.

In common, `int32_t` and int, as well as `uint32_t` and unsigned int, are different types.

If users do not make the aforementioned updates to format strings in their applications, the following error will be reported during compilation:
Removing CONFIG_COMPILER_DISABLE_GCC8_WARNINGS Build Option

CONFIG_COMPILER_DISABLE_GCC8_WARNINGS option was introduced to allow building of legacy code dating from the rigid GCC 5 toolchain. However, enough time has passed to allow for the warnings to be fixed, thus this option has been removed.

For now in GCC 11, users are advised to review their code and fix the compiler warnings where possible.

Networking

Wi-Fi

Callback Function Type esp_now_recv_cb_t  Previously, the first parameter of esp_now_recv_cb_t was of type const uint8_t *mac_addr, which only included the address of ESP-NOW peer device.

This now changes. The first parameter is of type esp_now_recv_info_t, which has members src_addr, des_addr and rx_ctrl. Therefore, the following updates are required:

- Redefine ESP-NOW receive callback function.
- src_addr can be used to replace original mac_addr.
- des_addr is the destination MAC address of ESP-NOW packet, which can be unicast or broadcast address.
  With des_addr, the user can distinguish unicast and broadcast ESP-NOW packets where broadcast ESP-NOW packets can be non-encrypted even when encryption policy is configured for the ESP-NOW.
- rx_ctrl is Rx control info of the ESP-NOW packet, which provides more information about the packet.

Please refer to the ESP-NOW example: wifi/espnow/main/espnow_example_main.c

Ethernet

esp_eth_ioctl() API  Previously, the esp_eth_ioctl() API had the following issues:

- The third parameter (which is of type void *) would accept an int/bool type arguments (i.e., not pointers) as input in some cases. However, these cases were not documented properly.
- To pass int/bool type argument as the third parameter, the argument had to be “unnaturally” casted to a void * type, to prevent a compiler warning as demonstrated in the code snippet below. This casting could lead to misuse of the esp_eth_ioctl() function.

```c
esp_eth_ioctl(eth_handle, ETH_CMD_S_FLOW_CTRL, (void *)true);
```

Therefore, the usage of esp_eth_ioctl() is now unified. Arguments to the third parameter must be passed as pointers to a specific data type to/from where data is stored/read by esp_eth_ioctl(). The code snippets below demonstrate the usage of esp_eth_ioctl().

Usage example to set Ethernet configuration:
eth_duplex_t new_duplex_mode = ETH_DUPLEX_HALF;
esp_eth_ioctl(eth_handle, ETH_CMD_S_DUPLEX_MODE, &new_duplex_mode);

Usage example to get Ethernet configuration:

eth_duplex_t duplex_mode;
esp_eth_ioctl(eth_handle, ETH_CMD_G_DUPLEX_MODE, &duplex_mode);

KSZ8041/81 and LAN8720 Driver Update  The KSZ8041/81 and LAN8720 drivers are updated to support more devices (i.e., generations) from their associated product families. The drivers can recognize particular chip numbers and their potential support by the driver.

As a result, the specific “chip number” functions calls are replaced by generic ones as follows:

- Removed esp_eth_phy_new_ksz8041() and esp_eth_phy_new_ksz8081(), and use esp_eth_phy_new_ksz80xx() instead
- Removed esp_eth_phy_new_lan8720(), and use esp_eth_phy_new_lan87xx() instead

ESP NETIF Glue Event Handlers  esp_eth_set_default_handlers() and esp_eth_clear_default_handlers() functions are removed. Registration of the default IP layer handlers for Ethernet is now handled automatically. If you have already followed the suggestion to fully initialize the Ethernet driver and network interface before registering their Ethernet/IP event handlers, then no action is required (except for deleting the affected functions). Otherwise, you may start the Ethernet driver right after they register the user event handler.

PHY Address Auto-detect  The Ethernet PHY address auto-detect function esp_eth_detect_phy_addr() is renamed to esp_eth_phy_802_3_detect_phy_addr() and its header declaration is moved to esp_eth/include/esp_eth_phy_802_3.h.

SPI-Ethernet Module Initialization  The SPI-Ethernet Module initialization is now simplified. Previously, you had to manually allocate an SPI device using spi_bus_add_device() before instantiating the SPI-Ethernet MAC.

Now, you no longer need to call spi_bus_add_device() as SPI devices are allocated internally. As a result, the eth_dm9051_config_t, eth_w5500_config_t, and eth_ksz8851snl_config_t configuration structures are updated to include members for SPI device configuration (e.g., to allow fine tuning of SPI timing which may be dependent on PCB design). Likewise, the ETH_DM9051_DEFAULT_CONFIG, ETH_W5500_DEFAULT_CONFIG, and ETH_KSZ8851SNL_DEFAULT_CONFIG configuration initialization macros are updated to accept new input parameters. Refer to Ethernet API Reference Guide for an example of SPI-Ethernet Module initialization.

Ethernet Driver  APIs for creating MAC instances (esp_eth_mac_new_*()) have been reworked to accept two parameters, instead of one common configuration. Now, the configuration includes

- Vendor specific MAC configuration
- Ethernet driver MAC configuration

This is applicable to internal Ethernet MAC esp_eth_mac_new_esp32() as well as to external MAC devices, such as esp_eth_mac_new_ksz8851snl(), esp_eth_mac_new_dm9051(), and esp_eth_mac_new_w5500()

TCP/IP Adapter  The TCP/IP Adapter was a network interface abstraction component used in ESP-IDF prior to v4.1. This section outlines migration from tcpip_adapter API to its successor ESP-NETIF.

Updating Network Connection Code
Network Stack Initialization

- You may simply replace `tcpip_adapter_init()` with `esp_netif_init()`. However, please note that the `esp_netif_init()` function now returns standard error codes. See *ESP-NETIF* for more details.
- The `esp_netif_deinit()` function is provided to de-initialize the network stack.
- You should also replace `#include "tcpip_adapter.h"` with `#include "esp_netif.h"`.

Network Interface Creation

Previously, the TCP/IP Adapter defined the following network interfaces statically:

- WiFi Station
- WiFi Access Point
- Ethernet

This now changes. Network interface instance should be explicitly constructed, so that the *ESP-NETIF* can connect to the TCP/IP stack. For example, after the TCP/IP stack and the event loop are initialized, the initialization code for WiFi must explicitly call `esp_netif_create_default_wifi_sta()`, or `esp_netif_create_default_wifi_ap()`.

Please refer to the 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`

Other tcpip_adapter API Replacement

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, as all events should be handled in the same way. Please note that for IP-related event handlers, application code usually receives IP addresses in the form of an `esp_netif` specific struct instead of the LwIP structs. However, both structs are binary compatible.

This is the preferred way to print the address:

```c
ESP_LOGI(TAG, "got ip:", IPSTR, IP2STR(&event->ip_info.ip));
```

Instead of

```c
ESP_LOGI(TAG, "got ip:%s", ip4addr_ntoa(&event->ip_info.ip));
```

Since `ip4addr_ntoa()` is a LwIP API, the `esp-netif` provides `esp_ip4addr_ntoa()` as a replacement. However, the above method using `IP2STR()` is generally preferred.

IP Addresses

You are advised to use `esp-netif` defined IP structures. Please note that with default compatibility enabled, the LwIP structs still work.

- `esp-netif` IP address definitions

Peripherals
Peripheral Clock Gating As usual, peripheral clock gating is still handled by driver itself, users do not need to take care of the peripheral module clock gating.

However, for advanced users who implement their own drivers based on hal and soc components, the previous clock gating include path has been changed from driver/periph_ctrl.h to esp_private/periph_ctrl.h.

RTC Subsystem Control RTC control APIs have been moved from driver/rtc_cntl.h to esp_private/rtc_ctrl.h.

ADC

ADC Oneshot & Continuous Mode Drivers The ADC oneshot mode driver has been redesigned.

- The new driver is in esp_adc component and the include path is esp_adc/adc_oneshot.h.
- The legacy driver is still available in the previous include path driver/adc.h.

The ADC continuous mode driver has been moved from driver component to esp_adc component.

- The include path has been changed from driver/adc.h to esp_adc/adc_continuous.h.

 Attempting to use the legacy include path driver/adc.h of either driver triggers the build warning below by default. However, the warning can be suppressed by enabling the CONFIG_ADC_SUPPRESS_DEPRECATE_WARN Kconfig option.

| legacy adc driver is deprecated, please migrate to use esp_adc/adc_oneshot.h and esp_adc/adc_continuous.h for oneshot mode and continuous mode drivers |

ADC Calibration Driver The ADC calibration driver has been redesigned.

- The new driver is in esp_adc component and the include path is esp_adc/adc_cali.h and esp_adc/adc_cali_scheme.h.

Legacy driver is still available by including esp_adc_cal.h. However, if users still would like to use the include path of the legacy driver, users should add esp_adc component to the list of component requirements in CMakeLists.txt.

 Attempting to use the legacy include path esp_adc_cal.h triggers the build warning below by default. However, the warning can be suppressed by enabling the CONFIG_ADC_CALI_SUPPRESS_DEPRECATE_WARN Kconfig option.

| legacy adc calibration driver is deprecated, please migrate to use esp_adc/adc_cali.h and esp_adc/adc_cali_scheme.h |

API Changes

- The ADC power management APIs adc_power_acquire and adc_power_release have made private and moved to esp_private/adc_share_hw_ctrl.h.
  - The two APIs were previously made public due to a HW errata workaround.
  - Now, ADC power management is completely handled internally by drivers.
  - Users who still require this API can include esp_private/adc_share_hw_ctrl.h to continue using these functions.

- driver/adc2_wifi_private.h has been moved to esp_private/adc_share_hw_ctrl.h.

- Enums ADC_UNIT_BOTH, ADC_UNIT_ALTER, and ADC_UNIT_MAX in adc_unit_t have been removed.

- The following enumerations have been removed as some of their enumeration values are not supported on all chips. This would lead to the driver triggering a runtime error if an unsupported value is used.
  - Enum ADC_CHANNEL_MAX
  - Enum ADC_ATTEN_MAX
– Enum ADC_CONV_UNIT_MAX

- API hall_sensor_read on ESP32 has been removed. Hall sensor is no longer supported on ESP32.
- API adc_set_i2s_data_source and adc_i2s_mode_init have been deprecated. Related enum adc_i2s_source_t has been deprecated. Please migrate to use esp_adc/adc_continuous.h.
- API adc_digi_filter_reset, adc_digi_filter_set_config, adc_digi_filter_get_config and adc_digi_filter_enable have been removed. These APIs behaviours are not guaranteed. Enum adc_digi_filter_idx_t, adc_digi_filter_mode_t and structure adc_digi_iir_filter_t have been removed as well.
- API esp_adc_cal_characterize has been deprecated, please migrate to adc_cali_create_scheme_curve_fitting or adc_cali_create_scheme_line_fitting instead.
- API esp_adc_cal_raw_to_voltage has been deprecated, please migrate to adc_cali_raw_to_voltage instead.
- API esp_adc_cal_get_voltage has been deprecated, please migrate to adc_oneshot_get_calibrated_result instead.

GPIO

- The previous Kconfig option RTCIO_SUPPORT_RTC_GPIO_DESC has been removed, thus the rtc_gpio_desc array is unavailable. Please use rtc_io_desc array instead.
- The user callback of a GPIO interrupt should no longer read the GPIO interrupt status register to get the GPIO’s pin number of triggering the interrupt. You should use the callback argument to determine the GPIO’s pin number instead.
  - Previously, when a GPIO interrupt occurs, the GPIO’s interrupt status register is cleared after calling the user callbacks. Thus, it was possible for users to read the GPIO’s interrupt status register inside the callback to determine which GPIO was used to trigger the interrupt.
  - However, clearing the interrupt status register after calling the user callbacks can potentially cause edge-triggered interrupts to be lost. For example, if an edge-triggered interrupt is triggered/retriggered while the user callbacks are being called, that interrupt will be cleared without its registered user callback being handled.
  - Now, the GPIO’s interrupt status register is cleared before invoking the user callbacks. Thus, users can no longer read the GPIO interrupt status register to determine which pin has triggered the interrupt. Instead, users should use the callback argument to pass the pin number.

Sigma-Delta Modulator  The Sigma-Delta Modulator driver has been redesigned into SDM.

- The new driver implements a factory pattern, where the SDM channels are managed in a pool internally, thus users do not have to fix a SDM channel to a GPIO manually.
- All SDM channels can be allocated dynamically.

Although it is recommended to use the new driver APIs, the legacy driver is still available in the previous include path driver/sigmadelta.h. However, by default, including driver/sigmadelta.h triggers the build warning below. The warning can be suppressed by Kconfig option CONFIG_SDM_SUPPRESS_DEPRECATED.Warn.

The legacy sigma-delta driver is deprecated, please use driver/sdm.h

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- SDM channel representation has changed from sigmadelta_channel_t to sdm_channel_handle_t, which is an opaque pointer.
- SDM channel configurations are stored in sdm_config_t now, instead the previous sigmadelta_config_t.
- In the legacy driver, users do not have to set the clock source for SDM channel. But in the new driver, users need to set a proper one in the sdm_config_t::clk_src. The available clock sources are listed in the soc_periph_sdm_clk_src_t.
• In the legacy driver, users need to set a prescale for the channel, which reflects the frequency in which the modulator outputs a pulse. In the new driver, users should use `sdm_config_t::sample_rate_hz` to set the over sample rate.

• In the legacy driver, users set duty to decide the output analog value, it is now renamed to a more appropriate name density.

### Breaking Changes in Usage

• Channel configuration was done by channel allocation, in `sdm_new_channel()`. In the new driver, only the density can be changed at runtime, by `sdm_channel_set_pulse_density()`. Other parameters like gpio number and prescale are only allowed to set during channel allocation.

• Before further channel operations, users should enable the channel in advance, by calling `sdm_channel_enable()`. This function helps to manage some system level services, like **Power Management**.

### Timer Group Driver

Timer Group driver has been redesigned into **GPTimer**, which aims to unify and simplify the usage of general purpose timer.

Although it is recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/timer.h`. However, by default, including `driver/timer.h` triggers the build warning below. The warning can be suppressed by the Kconfig option `CONFIG_GPTIMER_SUPPRESS_DEPRECATE_WARN`.

```
legacy timer group driver is deprecated, please migrate to driver/gptimer.h
```

The major breaking changes in concept and usage are listed as follows:

### Breaking Changes in Concepts

• `timer_group_t` and `timer_idx_t` which used to identify the hardware timer are removed from user’s code. In the new driver, a timer is represented by `gptimer_handle_t`.

• Definition of timer clock source is moved to `gptimer_clock_source_t`, the previous `timer_src_clk_t` is not used.

• Definition of timer count direction is moved to `gptimer_count_direction_t`, the previous `timer_count_dir_t` is not used.

• Only level interrupt is supported, `timer_intr_t` and `timer_intr_mode_t` are not used.

• Auto-reload is enabled by set the `gptimer_alarm_config_t::auto_reload_on_alarm` flag. `timer_autoreload_t` is not used.

### Breaking Changes in Usage

• Timer initialization is done by creating a timer instance from `gptimer_new_timer()`. Basic configurations like clock source, resolution and direction should be set in `gptimer_config_t`. Note that, specific configurations of alarm events are not needed during the installation stage of the driver.

• Alarm event is configured by `gptimer_set_alarm_action()`, with parameters set in the `gptimer_alarm_config_t`.

• Setting and getting count value are done by `gptimer_get_raw_count()` and `gptimer_set_raw_count()`. The driver does not help convert the raw value into UTC time-stamp. Instead, the conversion should be done from user’s side as the timer resolution is also known to the user.

• The driver will install the interrupt service as well if `gptimer_event_callbacks_t::on_alarm` is set to a valid callback function. In the callback, users do not have to deal with the low level registers (like “clear interrupt status”, “re-enable alarm event” and so on). So functions like `timer_group_get_intr_status_in_isr` and `timer_group_get_auto_reload_in_isr` are not used anymore.

• To update the alarm configurations when alarm event happens, one can call `gptimer_set_alarm_action()` in the interrupt callback, then the alarm will be re-enabled again.

• Alarm will always be re-enabled by the driver if `gptimer_alarm_config_t::auto_reload_on_alarm` is set to true.
Chapter 6. Migration Guides

### UART

<table>
<thead>
<tr>
<th>Removed/Deprecated items</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>uart_isr_register()</td>
<td>None</td>
<td>UART interrupt handling is implemented by driver itself.</td>
</tr>
<tr>
<td>uart_isr_free()</td>
<td>None</td>
<td>UART interrupt handling is implemented by driver itself.</td>
</tr>
<tr>
<td>use_ref_tick in uart_config_t</td>
<td>uart_enable_pattern_det_intr()</td>
<td>Select the clock source.</td>
</tr>
</tbody>
</table>

### I2C

<table>
<thead>
<tr>
<th>Removed/Deprecated items</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2c_isr_register()</td>
<td>None</td>
<td>I2C interrupt handling is implemented by driver itself.</td>
</tr>
<tr>
<td>i2c_isr_register()</td>
<td>None</td>
<td>I2C interrupt handling is implemented by driver itself.</td>
</tr>
<tr>
<td>i2c_opmode_t</td>
<td>None</td>
<td>It is not used anywhere in ESP-IDF.</td>
</tr>
</tbody>
</table>

### SPI

<table>
<thead>
<tr>
<th>Removed/Deprecated items</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>spi_cal_clock()</td>
<td>spi_get_actual_clock()</td>
<td>Get SPI real working frequency.</td>
</tr>
</tbody>
</table>

* The internal header file `spi_common_internal.h` has been moved to `esp_private/spi_common_internal.h`.

### LEDC

<table>
<thead>
<tr>
<th>Removed/Deprecated items</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit_num in ledc_timer_config_t</td>
<td>ledc_timer_config_t::duty_resolution</td>
<td>Set resolution of the duty cycle.</td>
</tr>
</tbody>
</table>

### Pulse Counter Driver

Pulse counter driver has been redesigned (see PCNT), which aims to unify and simplify the usage of PCNT peripheral.

Although it is recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/pcnt.h`. However, including `driver/pcnt.h` triggers the build warning below by default. The warning can be suppressed by the Kconfig option `CONFIG_PCNT_SUPPRESS_DEPRECATE_WARN`.

legacy pcnt driver is deprecated, please migrate to use driver/pulse_cnt.h

The major breaking changes in concept and usage are listed as follows:

#### Breaking Changes in Concepts

- `pcnt_port_t`, `pcnt_unit_t` and `pcnt_channel_t` which used to identify the hardware unit and channel are removed from user’s code. In the new driver, PCNT unit is represented by `pcnt_unit_handle_t`, likewise, PCNT channel is represented by `pcnt_channel_handle_t`. Both of them are opaque pointers.
- `pcnt_evt_type_t` is not used any more, they have been replaced by a universal **Watch Point Event**. In the event callback `pcnt_watch_cb_t`, it is still possible to distinguish different watch points from `pcnt_watch_event_data_t`.
- `pcnt_count_mode_t` is replaced by `pcnt_channel_edge_action_t`, and `pcnt_ctrl_mode_t` is replaced by `pcnt_channel_level_action_t`.
Chapter 6. Migration Guides

Breaking Changes in Usage

• Previously, the PCNT unit configuration and channel configuration were combined into a single function: `pcnt_unit_config`. They are now split into the two factory APIs: `pcnt_new_unit()` and `pcnt_new_channel()` respectively.
  – Only the count range is necessary for initializing a PCNT unit. GPIO number assignment has been moved to `pcnt_new_channel()`.
  – High/Low control mode and positive/negative edge count mode are set by stand-alone functions: `pcnt_channel_set_edge_action()` and `pcnt_channel_set_level_action()`.

• `pcnt_get_counter_value` is replaced by `pcnt_unit_get_count()`.

• `pcnt_counter_pause` is replaced by `pcnt_unit_stop()`.

• `pcnt_counter_resume` is replaced by `pcnt_unit_start()`.

• `pcnt_counter_clear` is replaced by `pcnt_unit_clear_count()`.

• `pcnt_intr_enable` and `pcnt_intr_disable` are removed. In the new driver, the interrupt is enabled by registering event callbacks `pcnt_unit_register_event_callbacks()`.

• `pcnt_event_enable` and `pcnt_event_disable` are removed. In the new driver, the PCNT events are enabled/disabled by adding/removing watch points `pcnt_unit_add_watch_point()`, `pcnt_unit_remove_watch_point()`.

• `pcnt_set_event_value` is removed. In the new driver, event value is also set when adding watch point by `pcnt_unit_add_watch_point()`.

• `pcnt_get_event_value` and `pcnt_get_event_status` are removed. In the new driver, these information are provided by event callback `pcnt_watch_cb_t` in the `pcnt_watch_event_data_t`.

• `pcnt_isr_register` and `pcnt_isr_unregister` are removed. Register of the ISR handler from user’s code is no longer permitted. Users should register event callbacks instead by calling `pcnt_unit_register_event_callbacks()`.

• `pcnt_set_mode` is replaced by `pcnt_channel_set_edge_action()` and `pcnt_channel_set_level_action()`.

• `pcnt_isr_service_install`, `pcnt_isr_service_uninstall`, `pcnt_isr_handler_add` and `pcnt_isr_handler_remove` are replaced by `pcnt_unit_register_event_callbacks()`. The default ISR handler is lazy installed in the new driver.

Temperature Sensor Driver  The temperature sensor driver has been redesigned and it is recommended to use the new driver. However, the old driver is still available but cannot be used with the new driver simultaneously.

The new driver can be included via `driver/temperature_sensor.h`. The old driver is still available in the previous include path `driver/temp_sensor.h`. However, including `driver/temp_sensor.h` triggers the build warning below by default. The warning can be suppressed by enabling the menuconfig option `CONFIG_TEMP_SENSOR_SUPPRESS_DEPRECATED_WARN`.

```
legacy temperature sensor driver is deprecated, please migrate to driver/
    temperature_sensor.h
```

Configuration contents has been changed. In the old version, users need to configure `clk_div` and `dac_offset`. While in the new version, users only need to choose `tsens_range`.

The process of using temperature sensor has been changed. In the old version, users can use `config->start->read_celsius` to get value. In the new version, users should install the temperature sensor driver firstly, by `temperature_sensor_install` and un-install it when finished. For more information, please refer to Temperature Sensor .

RMT Driver  RMT driver has been redesigned (see RMT transceiver), which aims to unify and extend the usage of RMT peripheral.
Although it is recommended to use the new driver APIs, the legacy driver is still available in the previous include path driver/rmt.h. However, including driver/rmt.h triggers the build warning below by default. The warning can be suppressed by the Kconfig option CONFIG_RMT_SUPPRESS_DEPRECATE_WARN.

The legacy RMT driver is deprecated, please use driver/rmt_tx.h and/or driver/rmt_rx.h

The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- **rmt_channel_t** which used to identify the hardware channel are removed from user space. In the new driver, RMT channel is represented by **rmt_channel_handle_t**. The channel is dynamically allocated by the driver, instead of designated by user.
- **rmt_item32_t** is replaced by **rmt_symbol_word_t**, which avoids a nested union inside a struct.
- **rmt_mem_t** is removed, as we do not allow users to access RMT memory block (a.k.a. an RMTMEM) directly. Direct access to RMTMEM does not make sense but make mistakes, especially when the RMT channel also connected with a DMA channel.
- **rmt_mem_owner_t** is removed, as the ownership is controlled by driver, not by user anymore.
- **rmt_source_clk_t** is replaced by **rmt_clock_source_t**, and note they are not binary compatible.
- **rmt_data_mode_t** is removed, the RMT memory access mode is configured to always use Non-FIFO and DMA mode.
- **rmt_mode_t** is removed, as the driver has stand alone install functions for TX and RX channels.
- **rmt_idle_level_t** is removed, setting IDLE level for TX channel is available in **rmt_transmit_config_t::eot_level**.
- **rmt_carrier_level_t** is removed, setting carrier polarity is available in **rmt_carrier_config_t::polarity_active_low**.
- **rmt_channel_status_t** and **rmt_channel_status_result_t** are removed, they are not used anywhere.
- Transmitting by RMT channel does not expect user to prepare the RMT symbols, instead, user needs to provide an RMT Encoder to tell the driver how to convert user data into RMT symbols.

Breaking Changes in Usage

- Channel installation has been separated for TX and RX channels into **rmt_new_tx_channel()** and **rmt_new_rx_channel()**.
- **rmt_set_clk_div** and **rmt_get_clk_div** are removed. Channel clock configuration can only be done during channel installation.
- **rmt_set_rx_idle_thresh** and **rmt_get_rx_idle_thresh** are removed. In the new driver, the RX channel IDLE threshold is redesigned into a new concept **rmt_receive_config_t::signal_range_max_ns**.
- **rmt_set_mem_block_num** and **rmt_get_mem_block_num** are removed. In the new driver, the memory block number is determined by **rmt_tx_channel_config_t::mem_block_symbols** and **rmt_rx_channel_config_t::mem_block_symbols**.
- **rmt_set_tx_carrier** is removed, the new driver uses **rmt_apply_carrier()** to set carrier behavior.
- **rmt_set_mem_pd** and **rmt_get_mem_pd** are removed. The memory power is managed by the driver automatically.
- **rmt_memory_rw_rst**, **rmt_tx_memory_reset** and **rmt_rx_memory_reset** are removed. Memory reset is managed by the driver automatically.
- **rmt_tx_start** and **rmt_rx_start** are merged into a single function **rmt_enable()**, for both TX and RX channels.
- **rmt_tx_stop** and **rmt_rx_stop** are merged into a single function **rmt_disable()**, for both TX and RX channels.
- **rmt_set_memory_owner** and **rmt_get_memory_owner** are removed. RMT memory owner guard is added automatically by the driver.
- **rmt_set_tx_loop_mode** and **rmt_get_tx_loop_mode** are removed. In the new driver, the loop mode is configured in **rmt_transmit_config_t::loop_count**.
• `rmt_set_source_clk` and `rmt_get_source_clk` are removed. Configuring clock source is only possible during channel installation by `rmt_tx_channel_config_t::clk_src` and `rmt_rx_channel_config_t::clk_src`.

• `rmt_set_rx_filter` is removed. In the new driver, the filter threshold is redesigned into a new concept `rmt_receive_config_t::signal_range_min_ns`.

• `rmt_set_idle_level` and `rmt_get_idle_level` are removed. Setting IDLE level for TX channel is available in `rmt_transmit_config_t::eot_level`.

• `rmt_set_rx_intr_en`, `rmt_set_err_intr_en`, `rmt_set_tx_intr_en`, `rmt_set_tx_thr_intr_en` and `rmt_set_rx_thr_intr_en` are removed. The new driver does not allow user to turn on/off interrupt from user space. Instead, it provides callback functions.

• `rmt_set_gpio` and `rmt_set_pin` are removed. The new driver does not support to switch GPIO dynamically at runtime.

• `rmt_config` is removed. In the new driver, basic configuration is done during the channel installation stage.

• `rmt_isr_register` and `rmt_isr_deregister` are removed, the interrupt is allocated by the driver itself.

• `rmt_driver_install` is replaced by `rmt_new_tx_channel()` and `rmt_new_rx_channel()`.

• `rmt_driver_uninstall` is replaced by `rmt_del_channel()`.

• `rmt_fill_tx_items`, `rmt_write_items` and `rmt_write_sample` are removed. In the new driver, user needs to provide an encoder to “translate” the user data into RMT symbols.

• `rmt_get_counter_clock` is removed, as the channel clock resolution is configured by user from `rmt_tx_channel_config_t::resolution_hz`.

• `rmt_wait_tx_done` is replaced by `rmt_tx_wait_all_done()`.

• `rmt_translator_init`, `rmt_translator_set_context` and `rmt_translator_get_context` are removed. In the new driver, the translator has been replaced by the RMT encoder.

• `rmt_get_ringbuf_handle` is removed. The new driver does not use Ringbuffer to save RMT symbols. Instead, the incoming data are saved to the user provided buffer directly. The user buffer can even be mounted to DMA link internally.

• `rmt_register_tx_end_callback` is replaced by `rmt_tx_register_event_callbacks()`, where user can register `rmt_tx_event_callbacks_t::on_trans_done` event callback.

• `rmt_set_intr_enable_mask` and `rmt_clr_intr_enable_mask` are removed, as the interrupt is handled by the driver, user does not need to take care of it.

• `rmt_add_channel_to_group` and `rmt_remove_channel_from_group` are replaced by RMT sync manager. Please refer to `rmt_new_sync_manager()`.

• `rmt_set_tx_loop_count` is removed. The loop count in the new driver is configured in `rmt_transmit_config_t::loop_count`.

• `rmt_enable_tx_loop_autostop` is removed. In the new driver, TX loop auto stop is always enabled if available, it is not configurable anymore.

**LCD**

• The LCD panel initialization flow is slightly changed. Now the `esp_lcd_panel_init()` will not turn on the display automatically. User needs to call `esp_lcd_panel_disp_on_off()` to manually turn on the display. Note, this is different from turning on backlight. With this breaking change, user can flash a predefined pattern to the screen before turning on the screen. This can help avoid random noise on the screen after a power on reset.

• `esp_lcd_panel_disp_off()` is deprecated, please use `esp_lcd_panel_disp_on_off()` instead.

• `dc_as_cmd_phase` is removed. The SPI LCD driver currently does not support a 9-bit SPI LCD. Please always use a dedicated GPIO to control the LCD D/C line.

• The way to register RGB panel event callbacks has been moved from the `esp_lcd_rgb_panel_config_t` into a separate API `esp_lcd_rgb_panel_register_event_callbacks()`. However, the event callback signature is not changed.

• Previous `relax_on_idle` flag in `esp_lcd_rgb_panel_config_t` has been renamed into `esp_lcd_rgb_panel_config_t::refresh_on_demand`, which expresses the same meaning but with a clear name.

• If the RGB LCD is created with the `refresh_on_demand` flag enabled, the driver will
not start a refresh in the `esp_lcd_panel_draw_bitmap()`. Now users have to call `esp_lcd_rgb_panel_refresh()` to refresh the screen by themselves.

• `esp_lcd_color_space_t` is deprecated, please use `lcd_color_space_t` to describe the color space, and use `lcd_rgb_element_order_t` to describe the data order of RGB color.

**Dedicated GPIO Driver**

• All of the dedicated GPIO related Low Level (LL) functions in `cpu_ll.h` have been moved to `dedic_gpio_cpu_ll.h` and renamed.

**I2S Driver** The I2S driver has been redesigned (see **I2S Driver**), which aims to rectify the shortcomings of the driver that were exposed when supporting all the new features of ESP32-C3 & ESP32-S3. The new driver’s APIs are available by including corresponding I2S mode’s header files `driver/i2s/include/driver/i2s_std.h`, `driver/i2s/include/driver/i2s_pdm.h`, or `driver/i2s/include/driver/i2s_tdm.h`.

Meanwhile, the old driver’s APIs in `driver/deprecated/driver/i2s.h` are still supported for backward compatibility. But there will be warnings if users keep using the old APIs in their projects, these warnings can be suppressed by the Kconfig option `CONFIG_I2S_SUPPRESS_DEPRECATE_WARN`.

Here is the general overview of the current I2S files:

```
```

**Breaking changes in Concepts**

**Independent TX/RX channels** The minimum control unit in new I2S driver are now individual TX/RX channels instead of an entire I2S controller (that consists of multiple channels).

• The TX and RX channels of the same I2S controller can be controlled separately, meaning that they are configured such that they can be started or stopped separately.
• The `i2s_chan_handle_t` handle type is used to uniquely identify I2S channels. All the APIs require the channel handle and users need to maintain the channel handles by themselves.
• On the ESP32-C3 and ESP32-S3, TX and RX channels in the same controller can be configured to different clocks or modes.
• However, on the ESP32 and ESP32-S2, the TX and RX channels of the same controller still share some hardware resources. Thus, configurations may cause one channel to affect another channel in the same controller.
• The channels can be registered to an available I2S controller automatically by setting `i2s_port_t::I2S_NUM_AUTO` as I2S port ID which causes the driver to search for the available TX/RX channels. However, the driver also supports registering channels to a specific port.
• In order to distinguish between TX/RX channels and sound channels, the term “channel” in the context of the I2S driver only refers to TX/RX channels. Meanwhile, sound channels are referred to as “slots”.

**I2S Mode Categorization**  I2S communication modes are categorized into the following three modes. Note that:

- **Standard mode**: Standard mode always has two slots, it can support Philips, MSB, and PCM (short frame sync) formats. Please refer to `driver/i2s/include/driver/i2s_std.h` for more details.
- **PDM mode**: PDM mode only supports two slots with 16-bit data width, but the configurations of PDM TX and PDM RX are slightly different. For PDM TX, the sample rate can be set by `i2s_pdm_tx_clk_config_t::sample_rate`, and its clock frequency depends on the up-sampling configuration. For PDM RX, the sample rate can be set by `i2s_pdm_rx_clk_config_t::sample_rate`, and its clock frequency depends on the down-sampling configuration. Please refer to `driver/i2s/include/driver/i2s_pdm.h` for details.
- **TDM mode**: TDM mode can support up to 16 slots. It can work in Philips, MSB, PCM (short frame sync), and PCM (long frame sync) formats. Please refer to `driver/i2s/include/driver/i2s_tdm.h` for details.

When allocating a new channel in a specific mode, users should initialize that channel by its corresponding function. It is strongly recommended to use the helper macros to generate the default configurations in case the default values are changed in the future.

**Independent Slot and Clock Configuration**  The slot configurations and clock configurations can be configured separately.

- Call `i2s_channel_init_std_mode()`, `i2s_channel_init_pdm_rx_mode()`, `i2s_channel_init_pdm_tx_mode()`, or `i2s_channel_init_tdm_mode()` to initialize the slot/clock/gpio_pin configurations.
- Calling `i2s_channel_reconfig_std_slot()`, `i2s_channel_reconfig_pdm_rx_slot()`, `i2s_channel_reconfig_pdm_tx_slot()`, or `i2s_channel_reconfig_tdm_slot()` can change the slot configurations after initialization.
- Calling `i2s_channel_reconfig_std_clock()`, `i2s_channel_reconfig_pdm_rx_clock()`, `i2s_channel_reconfig_pdm_tx_clock()`, or `i2s_channel_reconfig_tdm_clock()` can change the clock configurations after initialization.
- Calling `i2s_channel_reconfig_std_gpio()`, `i2s_channel_reconfig_pdm_rx_gpio()`, `i2s_channel_reconfig_pdm_tx_gpio()`, or `i2s_channel_reconfig_tdm_gpio()` can change the GPIO configurations after initialization.

**Misc**

- States and state-machine are adopted in the new I2S driver to avoid APIs called in wrong state.
- ADC and DAC modes are removed. They are only supported in their own drivers and the legacy I2S driver.

**Breaking Changes in Usage**  To use the new I2S driver, please follow these steps:

1. Call `i2s_new_channel()` to acquire channel handles. We should specify the work role and I2S port in this step. Besides, the TX or RX channel handle will be generated by the driver. Inputting both two TX and RX channel handles is not necessary but at least one handle is needed. In the case of inputting both two handles, the driver will work at the duplex mode. Both TX and RX channels will be available on a same port, and they will share the MCLK, BCLK and WS signal. But if only one of the TX or RX channel handle is inputted, this channel will only work in the simplex mode.
2. Call `i2s_channel_init_std_mode()`, `i2s_channel_init_pdm_rx_mode()`, `i2s_channel_init_pdm_tx_mode()` or `i2s_channel_init_tdm_mode()` to initialize the channel to the specified mode. Corresponding slot, clock and GPIO configurations are needed in this step.
3. (Optional) Call `i2s_channel_register_event_callback()` to register the ISR event callback functions. I2S events now can be received by the callback function synchronously, instead of from the event queue asynchronously.
4. Call `i2s_channel_enable()` to start the hardware of I2S channel. In the new driver, I2S does not start automatically after installed, and users are supposed to know clearly whether the channel has started or not.
5. Read or write data by `i2s_channel_read()` or `i2s_channel_write()`. Certainly, only the RX channel handle is supposed to be inputted in `i2s_channel_read()` and the TX channel handle in `i2s_channel_write()`.
6. (Optional) The slot, clock and GPIO configurations can be changed by corresponding `reconfig` functions, but `i2s_channel_disable()` must be called before updating the configurations.
7. Call `i2s_channel_disable()` to stop the hardware of I2S channel.
8. Call `i2s_del_channel()` to delete and release the resources of the channel if it is not needed any more, but the channel must be disabled before deleting it.

**TWAI Driver** The deprecated CAN peripheral driver is removed. Please use TWAI driver instead (i.e., include `driver/twai.h` in your application).

**Register Access Macros** Previously, all register access macros could be used as expressions, so the following was allowed:

```c
uint32_t val = REG_SET_BITS(reg, bits, mask);
```

In ESP-IDF v5.0, register access macros which write or read-modify-write the register can no longer be used as expressions, and can only be used as statements. This applies to the following macros: `REG_WRITE`, `REG_SET_BIT`, `REG_CLR_BIT`, `REG_SET_BITS`, `REG_SET_FIELD`, `WRITE_PERI_REG`, `CLEAR_PERI_REG_MASK`, `SET_PERI_REG_MASK`, `SET_PERI_REG_BITS`.

To store the value which would have been written into the register, split the operation as follows:

```c
uint32_t new_val = REG_READ(reg) | mask;
REG_WRITE(reg, new_val);
```

To get the value of the register after modification (which may be different from the value written), add an explicit read:

```c
REG_SET_BITS(reg, bits, mask);
uint32_t new_val = REG_READ(reg);
```

**Protocols**

**Mbed TLS** For ESP-IDF v5.0, Mbed TLS has been updated from v2.x to v3.1.0.

For more details about Mbed TLS’s migration from version 2.x to version 3.0 or greater, please refer to the official guide.

**Breaking Changes (Summary)**

**Most Structure Fields Are Now Private**

- Direct access to fields of structures (struct types) declared in public headers is no longer supported.
- Appropriate accessor functions (getter/setter) must be used for the same. A temporary workaround would be to use `MBEDTLS_PRIVATE` macro (not recommended).
- For more details, refer to the official guide.
Chapter 6. Migration Guides

SSL

- Removed support for TLS 1.0, 1.1, and DTLS 1.0
- Removed support for SSL 3.0

Deprecated Functions Were Removed from Cryptography Modules

- The functions `MBEDTLS_*_ret()` (related to MD, SHA, RIPEMD, RNG, HMAC modules) was renamed to replace the corresponding functions without `_ret` appended and updated return value.
- For more details, refer to the official guide.

Deprecated Config Options

Following are some of the important config options deprecated by this update. The configs related to and/or dependent on these have also been deprecated.

- `MBEDTLS_SSL_PROTO_SSL3`: Support for SSL 3.0
- `MBEDTLS_SSL_PROTO_TLS1`: Support for TLS 1.0
- `MBEDTLS_SSL_PROTO_TLS1_1`: Support for TLS 1.1
- `MBEDTLS_SSL_PROTO_DTLS`: Support for DTLS 1.1 (Only DTLS 1.2 is supported now)
- `MBEDTLS_DES_C`: Support for 3DES ciphersuites
- `MBEDTLS_RC4_MODE`: Support for RC4-based ciphersuites

Note: This list includes only major options configurable through `idf.py menuconfig`. For more details on deprecated options, refer to the official guide.

Miscellaneous

Disabled Diffie-Hellman Key Exchange Modes

The Diffie-Hellman Key Exchange modes have now been disabled by default due to security risks (see warning text here). Related configs are given below:

- `MBEDTLS_DHM_C`: Support for the Diffie-Hellman-Merkle module
- `MBEDTLS_KEY_EXCHANGE_DHE_PSK`: Support for Diffie-Hellman PSK (pre-shared-key) TLS authentication modes
- `MBEDTLS_KEY_EXCHANGE_DHE_RSA`: Support for cipher suites with the prefix `TLS-DHE-RSA-WITH-

Note: During the initial step of the handshake (i.e., `client_hello`), the server selects a cipher from the list that the client publishes. As the DHE_PSK/DHE_RSA ciphers have now been disabled by the above change, the server would fall back to an alternative cipher; if in a rare case, it does not support any other cipher, the handshake would fail. To retrieve the list of ciphers supported by the server, one must attempt to connect with the server with a specific cipher from the client-side. Few utilities can help do this, e.g., `sslscan`.

Remove certs Module from X509 Library

- The `MBEDTLS/certs.h` header is no longer available in `mbedtls 3.1`. Most applications can safely remove it from the list of includes.

Breaking Change for `esp_crt_bundle_set` API

- The `esp_crt_bundle_set()` API now requires one additional argument named `bundle_size`. The return type of the API has also been changed to `esp_err_t` from `void`.

Breaking Change for `esp_ds_rsa_sign` API

- The `esp_ds_rsa_sign()` API now requires one less argument. The argument `mode` is no longer required.
HTTPS Server

Breaking Changes (Summary)  Names of variables holding different certs in httpd_ssl_config_t structure have been updated.

- httpd_ssl_config::servercert variable inherits role of cacert_pem variable.
- httpd_ssl_config::servercert_len variable inherits role of cacert_len variable
- httpd_ssl_config::cacert_pem variable inherits role of client_verify_cert_pem variable
- httpd_ssl_config::cacert_len variable inherits role of client_verify_cert_len variable

The return type of the httpd_ssl_stop() API has been changed to esp_err_t from void.

ESP HTTPS OTA

Breaking Changes (Summary)

- The function esp_https_ota() now requires pointer to esp_https_ota_config_t as argument instead of pointer to esp_http_client_config_t.

ESP-TLS

Breaking Changes (Summary)

esp_tls_t Structure Is Now Private  The esp_tls_t has now been made completely private. You cannot access its internal structures directly. Any necessary data that needs to be obtained from the ESP-TLS handle can be done through respective getter/setter functions. If there is a requirement of a specific getter/setter function, please raise an issue on ESP-IDF.

The list of newly added getter/setter function is as as follows:

- esp_tls_get_ssl_context() - Obtain the ssl context of the underlying ssl stack from the ESP-TLS handle.

Function Deprecations And Recommended Alternatives  Following table summarizes the deprecated functions removed and their alternatives to be used from ESP-IDF v5.0 onwards.

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp_tls_conn_new()</td>
<td>esp_tls_conn_new_sync()</td>
</tr>
<tr>
<td>esp_tls_conn_delete()</td>
<td>esp_tls_conn_destroy()</td>
</tr>
</tbody>
</table>

- The function esp_tls_conn_http_new() has now been termed as deprecated. Please use the alternative function esp_tls_conn_http_new_sync() (or its asynchronous esp_tls_conn_http_new_async()). Note that the alternatives need an additional parameter esp_tls_t, which has to be initialized using the esp_tls_init() function.

HTTP Server

Breaking Changes (Summary)

- http_server.h header is no longer available in esp_http_server. Please use esp_http_server.h instead.
ESP HTTP Client

Breaking Changes (Summary)

- The functions `esp_http_client_read()` and `esp_http_client_fetch_headers()` now return an additional return value -ESP_ERR_HTTP_EAGAIN for timeout errors - call timed-out before any data was ready.

TCP Transport

Breaking Changes (Summary)

- The function `esp_transport_read()` now returns 0 for a connection timeout and < 0 for other errors. Please refer `esp_tcp_transport_err_t` for all possible return values.

MQTT Client

Breaking Changes (Summary)

- `esp_mqtt_client_config_t` have all fields grouped in sub structs.

Most common configurations are listed below:

- Broker address now is set in `esp_mqtt_client_config_t::broker::address::uri`
- Security related to broker verification in `esp_mqtt_client_config_t::broker::verification`
- Client username is set in `esp_mqtt_client_config_t::credentials::username`
- `esp_mqtt_client_config_t` no longer supports the `user_context` field. Please use `esp_mqtt_client_register_event()` instead for registering an event handler; the last argument `event_handler_arg` can be used to pass user context to the handler.

ESP-Modbus

Breaking Changes (Summary)  
The ESP-IDF component `freemodbus` has been removed from ESP-IDF and is supported as a separate component. Additional information for the ESP-Modbus component can be found in the separate repository:

- ESP-Modbus component on GitHub

The main component folder of the new application shall include the component manager manifest file `idf_component.yml` as in the example below:

```yaml
dependencies:
  espressif/sp-modbus:
    version: "^1.0"
```

The `sp-modbus` component can be found in component manager registry. Refer to component manager documentation for more information on how to set up the component manager.

For applications targeting v4.x releases of ESP-IDF that need to use new `sp-modbus` component, adding the component manager manifest file `idf_component.yml` will be sufficient to pull in the new component. However, users should also exclude the legacy `freemodbus` component from the build. This can be achieved using the statement below in the project’s `CMakeLists.txt`:

```cmake
set(EXCLUDE_COMPONENTS freemodbus)
```
Provisioning

**Protocomm**  The `pop` field in the `protocomm_set_security()` API is now deprecated. Please use the `sec_params` field instead of `pop`. This parameter should contain the structure (including the security parameters) as required by the protocol version used.

For example, when using security version 2, the `sec_params` parameter should contain the pointer to the structure of type `protocomm_security2_params_t`.

**Wi-Fi Provisioning**

- The `pop` field in the `wifi_prov_mgr_start_provisioning()` API is now deprecated. For backward compatibility, `pop` can be still passed as a string for security version 1. However, for security version 2, the `wifi_prov_sec_params` argument needs to be passed instead of `pop`. This parameter should contain the structure (containing the security parameters) as required by the protocol version used. For example, when using security version 2, the `wifi_prov_sec_params` parameter should contain the pointer to the structure of type `wifi_prov_security2_params_t`. For security 1, the behaviour and the usage of the API remain the same.

- The API `wifi_prov_mgr_is_provisioned()` does not return `ESP_ERR_INVALID_STATE` error any more. This API now works without any dependency on provisioning manager initialization state.

**ESP Local Control**  The `pop` field in the `esp_local_ctrl_proto_sec_cfg_t` API is now deprecated. Please use the `sec_params` field instead of `pop`. This field should contain the structure (containing the security parameters) as required by the protocol version used.

For example, when using security version 2, the `sec_params` field should contain pointer to the structure of type `esp_local_ctrl_security2_params_t`.

**Removed or Deprecated Components**

**Components Moved to ESP-IDF Component Registry**  Following components are removed from ESP-IDF and moved to ESP-IDF Component Registry:

- libsodium
- cbor
- jsmn
- esp_modem
- nghttp
- mdns
- esp_websocket_client
- asio
- freemodbus
- sh2lib
- expat
- coap
- esp-cryptoauthlib
- qrcode
- tjpgd
- esp_serial_slave_link
- tinyusb
Chapter 6. Migration Guides

**Note:** Please note that http parser functionality which was previously part of nghttp component is now part of http_parser component.

These components can be installed using idf.py add-dependency command.

For example, to install libsodium component with the exact version X.Y, run `idf.py add-dependency libsodium==X.Y`.

To install libsodium component with the latest version compatible to X.Y according to semver rules, run `idf.py add-dependency libsodium~X.Y`.

To find out which versions of each component are available, open https://components.espressif.com, search for the component by its name and check the versions listed on the component page.

**Deprecated Components** The following components are removed since they were deprecated in ESP-IDF v4.x:

- tcpip_adapter. Please use the ESP-NETIF component instead; you can follow the TCP/IP Adapter.

**Note:** OpenSSL-API component is no longer supported. It is not available in the IDF Component Registry, either. Please use ESP-TLS or mbedtls API directly.

**Note:** esp_adc_cal component is no longer supported. New adc calibration driver is in esp_adc component. Legacy adc calibration driver has been moved into esp_adc component. To use legacy esp_adc_cal driver APIs, you should add esp_adc component to the list of component requirements in CMakeLists.txt. Also check Peripherals Migration Guide for more details.

The targets components are no longer necessary after refactoring and have been removed:

- esp32
- esp32s2
- esp32s3
- esp32c2
- esp32c3
- esp32h2

**Storage**

**New Component for the Partition APIs** Breaking change: all the Partition API code has been moved to a new component esp_partition. For the complete list of affected functions and data-types, see header file esp_partition.h.

These API functions and data-types were previously a part of the spi_flash component, and thus possible dependencies on the spi_flash in existing applications may cause the build failure, in case of direct esp_partition_* APIs/data-types use (for instance, fatal error: esp_partition.h: No such file or directory at lines with #include "esp_partition.h"). If you encounter such an issue, please update your project’s CMakeLists.txt file as follows:

Original dependency setup:

```c
idf_component_register(...
    REQUIRES spi_flash)
```

Updated dependency setup:

```c
idf_component_register(...
    REQUIRES spi_flash esp_partition)
```
Note: Please update relevant `REQUIRES` or `PRIV_REQUIRES` section according to your project. The above-presented code snippet is just an example.

If the issue persists, please let us know and we will assist you with your code migration.

**SDMMC/SDSPI** SD card frequency on SDMMC/SDSPI interface can be now configured through `sdmmc_host_t.max_freq_khz` to a specific value, not only `SDMMC_FREQ_PROBING` (400 kHz), `SDMMC_FREQ_DEFAULT` (20 MHz), or `SDMMC_FREQ_HIGHSPEED` (40 MHz). Previously, in case you have specified a custom frequency other than any of the above-mentioned values, the closest lower-or-equal one was selected anyway.

Now, the underlying drivers calculate the nearest fitting value, given by available frequency dividers instead of an enumeration item selection. This could cause troubles in communication with your SD card without a change of the existing application code. If you encounter such an issue, please, keep trying different frequencies around your desired value unless you find the one working well. To check the frequency value calculated and actually applied, use `void sdmmc_card_print_info(FILE* stream, const sdmmc_card_t* card)` function.

**FatFs** FatFs is now updated to v0.14. As a result, the function signature of `f_mkfs()` has changed. The new signature is `FRESULT f_mkfs (const TCHAR* path, const MKFS_PARM* opt, void* work, UINT len)`; which uses `MKFS_PARM` struct as a second argument.

**Partition Table** The partition table generator no longer supports misaligned partitions. When generating a partition table, ESP-IDF only accepts partitions with offsets that align to 4 KB. This change only affects generating new partition tables. Reading and writing to already existing partitions remains unchanged.

**VFS** The `esp_vfs_semihost_register()` function signature is changed as follows:

- The new signature is `esp_err_t esp_vfs_semihost_register(const char* base_path);`
- The `host_path` parameter of the old signature no longer exists. Instead, the OpenOCD command `ESP_SEMIHOST_BASEDIR` should be used to set the full path on the host.

**Function Signature Changes** The following functions now return `esp_err_t` instead of `void` or `nvs_iterator_t`. Previously, when parameters were invalid or when something goes wrong internally, these functions would `assert()` or return a `nullptr`. With an `esp_err_t` returned, you can get better error reporting.

- `nvs_entry_find()`
- `nvs_entry_next()`  
- `nvs_entry_info()`

Because the `esp_err_t` return type changes, the usage patterns of `nvs_entry_find()` and `nvs_entry_next()` become different. Both functions now modify iterators via parameters instead of returning an iterator.

The old programming pattern to iterate over an NVS partition was as follows:

```c
nvs_iterator_t it = nvs_entry_find(<nvs_partition_name>, <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'", info.key, info.type);
}
```

The new programming pattern to iterate over an NVS partition is now:

```c
nvs_iterator_t it = nvs_entry_find(<nvs_partition_name>, <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'", info.key, info.type);
}
```
nvs_iterator_t it = nullptr;
esp_err_t res = nvs_entry_find(<nvs_partition_name>, <namespace>, NVS_TYPE_ANY, &it);
while(res == ESP_OK) {
    nvs_entry_info_t info;
    nvs_entry_info(it, &info); // Can omit error check if parameters are
    // guaranteed to be non-NULL
    printf("key '%s', type '%d'", info.key, info.type);
    res = nvs_entry_next(it);
} nvs_release_iterator(it);

**Iterator Validity**  Note that because the function signature changes, if there is a parameter error, you may get an invalid iterator from nvs_entry_find(). Hence, it is important to initialize the iterator to NULL before using nvs_entry_find(), so that you can avoid complex error checking before calling nvs_release_iterator(). A good example is the programming pattern above.

**Removed SDSPI Deprecated API** Structure sdspi_slot_config_t and function sdspi_host_init_slot() are removed, and replaced by structure sdspi_device_config_t and function sdspi_host_init_device() respectively.

**ROM SPI Flash** In versions before v5.0, ROM SPI flash functions were included via esp32**/rom/spi_flash.h. Thus, code written to support different ESP chips might be filled with ROM headers of different targets. Furthermore, not all of the APIs could be used on all ESP chips.

Now, the common APIs are extracted to esp_rom_spiflash.h. Although it is not a breaking change, you are strongly recommended to only use the functions from this header (i.e., prefixed with esp_rom_spiflash and included by esp_rom_spiflash.h) for better cross-compatibility between ESP chips.

To make ROM SPI flash APIs clearer, the following functions are also renamed:

- esp_rom_spiflash_lock() to esp_rom_spiflash_set_bp()
- esp_rom_spiflash_unlock() to esp_rom_spiflash_clear_bp()

**SPI Flash Driver**  The esp_flash_speed_t enum type is now deprecated. Instead, you may now directly pass the real clock frequency value to the flash configuration structure. The following example demonstrates how to configure a flash frequency of 80MHz:

```c
esp_flash_spi_device_config_t dev_cfg = {
    // Other members
    .freq_mhz = 80,
    // Other members
};
```

**Legacy SPI Flash Driver**  To make SPI flash drivers more stable, the legacy SPI flash driver is removed from v5.0. The legacy SPI flash driver refers to default spi_flash driver since v3.0, and the SPI flash driver with configuration option CONFIG_SPI_FLASH_USE_LEGACY_IMPL enabled since v4.0. The major breaking change here is that the legacy spi_flash driver is no longer supported from v5.0. Therefore, the legacy driver APIs and the CONFIG_SPI_FLASH_USE_LEGACY_IMPL configuration option are both removed. Please use the new spi_flash driver’s APIs instead.
<table>
<thead>
<tr>
<th>Removed Items</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spi_flash_erase_sector()</code></td>
<td><code>esp_flash_erase_region()</code></td>
</tr>
<tr>
<td><code>spi_flash_erase_range()</code></td>
<td><code>esp_flash_erase_region()</code></td>
</tr>
<tr>
<td><code>spi_flash_write()</code></td>
<td><code>esp_flash_write()</code></td>
</tr>
<tr>
<td><code>spi_flash_read()</code></td>
<td><code>esp_flash_read()</code></td>
</tr>
<tr>
<td><code>spi_flash_write_encrypted()</code></td>
<td><code>esp_flash_write_encrypted()</code></td>
</tr>
<tr>
<td><code>spi_flash_read_encrypted()</code></td>
<td><code>esp_flash_read_encrypted()</code></td>
</tr>
</tbody>
</table>

**Note:** New functions with prefix `esp_flash` accept an additional `esp_flash_t*` parameter. You can simply set it to NULL. This will make the function to run the main flash (`esp_flash_default_chip`).

The `esp_spi_flash.h` header is deprecated as system functions are no longer public. To use flash memory mapping APIs, you may include `spi_flash_mmap.h` instead.

**System**

**Inter-Processor Call**  IPC (Inter-Processor Call) feature is no longer a stand-alone component and has been integrated into the `esp_system` component.

Thus, any project presenting a `CMakeLists.txt` file with the parameters `PRIV_REQUIRES esp_ipc` or `REQUIRES esp_ipc` should be modified to simply remove these options as the `esp_system` component is included by default.

**ESP Clock**  The ESP Clock API (functions/types/macros prefixed with `esp_clk`) has been made into a private API. Thus, the previous include paths `#include "ESP32-S2/clk.h"` and `#include "esp_clk.h"` have been removed. If users still require usage of the ESP Clock API (though this is not recommended), it can be included via `#include "esp_private/esp_clk.h"`.

**Note:** Private APIs are not stable and no are longer subject to the ESP-IDF versioning scheme’s breaking change rules. Thus, it is not recommended for users to continue calling private APIs in their applications.

**Cache Error Interrupt**  The Cache Error Interrupt API (functions/types/macros prefixed with `esp_cache_err`) has been made into a private API. Thus, the previous include path `#include "ESP32-S2/cache_err_int.h"` has been removed. If users still require usage of the Cache Error Interrupt API (though this is not recommended), it can be included via `#include "esp_private/cache_err_int.h"`.

** BOOTLOADER SUPPORT**

- The function `bootloader_common_get_reset_reason()` has been removed. Please use the function `esp_rom_get_reset_reason()` in the ROM component.
- The functions `esp_secure_boot_verify_sbv2_signature_block()` and `esp_secure_boot_verify_rsa_signature_block()` have been removed without replacement. We do not expect users to use these directly. If they are indeed still necessary, please open a feature request on GitHub explaining why these functions are necessary to you.

**Brownout**  The Brownout API (functions/types/macros prefixed with `esp_brownout`) has been made into a private API. Thus, the previous include path `#include "brownout.h"` has been removed. If users still require usage of the Brownout API (though this is not recommended), it can be included via `#include "esp_private/brownout.h"`.

Submit Document Feedback
Trax The Trax API (functions/types/macros prefixed with `trax_`) has been made into a private API. Thus, the previous include path `#include "trax.h"` has been removed. If users still require usage of the Trax API (though this is not recommended), it can be included via `#include "esp_private/trax.h"`.

ROM The previously deprecated ROM-related header files located in `components/esp32/rom/` (old include path: `rom/*.h`) have been moved. Please use the new target-specific path from `components/esp_rom/include/ESP32-S2/` (new include path: `ESP32-S2/rom/*.h`).

**esp_hw_support**

- The header files `soc/cpu.h` have been deleted and deprecated CPU util functions have been removed. ESP-IDF developers should include `esp_cpu.h` instead for equivalent functions.
- The header files `hal/cpu_ll.h`, `hal/cpu_hal.h`, `hal/soc_ll.h`, `hal/soc_hal.h` and `interrupt_controller_hal.h` CPU API functions have been deprecated. ESP-IDF developers should include `esp_cpu.h` instead for equivalent functions.
- The header file `compare_set.h` have been deleted. ESP-IDF developers should use `esp_cpu_compare_and_set()` function provided in `esp_cpu.h` instead.
- `esp_cpu_get_ccount()`, `esp_cpu_set_ccount()` and `esp_cpu_in_ OCD_debug_mode()` were removed from `esp_cpu.h`. ESP-IDF developers should use respectively `esp_cpu_get_cycle_count()`, `esp_cpu_set_cycle_count()` and `esp_cpu_dbg_is_attached()` instead.
- The header file `esp_intr.h` has been deleted. Please include `esp_intr_alloc.h` to allocate and manipulate interrupts.
- The Panic API (functions/types/macros prefixed with `esp_panic`) has been made into a private API. Thus, the previous include path `#include "esp_panic.h"` has been removed. If users still require usage of the Trax API (though this is not recommended), it can be included via `#include "esp_private/panic_reason.h"`. Besides, developers should include `esp_debug_helpers.h` instead to use any debug-related helper functions, e.g., print backtrace.
- The header file `soc_log.h` is now renamed to `esp_hw_log.h` and has been made private. Users are encouraged to use logging APIs provided under `esp_log.h` instead.
- The header files `spinlock.h`, `clk_ctrl_os.h`, and `rtc_wdt.h` must now be included without the `soc` prefix. For example, `#include "spinlock.h"`.
- `esp_chip_info()` returns the chip version in the format = 100 * major eFuse version + minor eFuse version. Thus, the revision in the `esp_chip_info_t` structure is expanded to `uint16_t` to fit the new format.

**PSRAM**

- The target-specific header file `spiram.h` and the header file `esp_spiram.h` have been removed. A new component `esp_psram` is created instead. For PSRAM/SPIRAM-related functions, users now include `esp_psram.h` and set the `esp_psram` component as a component requirement in their `CMakeLists.txt` project files.
- `esp_spiram_get_chip_size` and `esp_spiram_get_size` have been deleted. You should use `esp_psram_get_size` instead.

**eFuse**

- The parameter type of function `esp_secure_boot_read_key_digests()` changed from `ets_secure_boot_key_digests_t*` to `esp_secure_boot_key_digests_t*`. The new type is the same as the old one, except that the `allow_key_revoke` flag has been removed. The latter was always set to `true` in current code, not providing additional information.
- Added eFuse wafer revisions: major and minor. The `esp_efuse_get_chip_ver()` API is not compatible with these changes, so it was removed. Instead, please use the following APIs: `efuse_hal_get_major_chip_version()`, `efuse_hal_get_minor_chip_version()` or `efuse_hal_chip_revision()`.
esp_common  EXT_RAM_ATTR is deprecated. Use the new macro EXT_RAM_BSS_ATTR to put .bss on PSRAM.

esp_system

- The header files esp_random.h, esp_mac.h, and esp_chip_info.h, which were all previously indirectly included via the header file esp_system.h, must now be included directly. These indirect inclusions from esp_system.h have been removed.
- The Backtrace Parser API (functions/types/macros prefixed with esp_eh_frame_) has been made into a private API. Thus, the previous include path #include "eh_frame_parser.h" has been removed. If users still require usage of the Backtrace Parser API (though this is not recommended), it can be included via #include "esp_private/eh_frame_parser.h".
- The Interrupt Watchdog API (functions/types/macros prefixed with esp_int_wdt_) has been made into a private API. Thus, the previous include path #include "esp_int_wdt.h" has been removed. If users still require usage of the Interrupt Watchdog API (though this is not recommended), it can be included via #include "esp_private/esp_int_wdt.h".

SoC Dependency

- Public API headers listed in the Doxyfiles will not expose unstable and unnecessary SoC header files, such as soc/soc.h and soc/rtc.h. That means the user has to explicitly include them in their code if these “missing” header files are still wanted.
- Kconfig option LEGACY_INCLUDE_COMMON_HEADERS is also removed.
- The header file soc/soc_memory_types.h has been deprecated. Users should use the esp_memory_utils.h instead. Including soc/soc_memory_types.h will bring a build warning like soc_memory_types.h is deprecated, please migrate to esp_memory_utils.h

APP Trace  One of the timestamp sources has changed from the legacy timer group driver to the new GPTimer. Kconfig choices like APPTRACE_SV_TS_SOURCE_TIMER00 has been changed to APPTRACE_SV_TS_SOURCE_GPTIMER. User no longer need to choose the group and timer ID.

esp_timer  The FRC2-based legacy implementation of esp_timer available on ESP32 has been removed. The simpler and more efficient implementation based on the LAC timer is now the only option.

ESP Image  The image SPI speed enum definitions have been renamed.

- Enum ESP_IMAGE_SPI_SPEED_80M has been renamed to ESP_IMAGE_SPI_SPEED_DIV_1.
- Enum ESP_IMAGE_SPI_SPEED_40M has been renamed to ESP_IMAGE_SPI_SPEED_DIV_2.
- Enum ESP_IMAGE_SPI_SPEED_26M has been renamed to ESP_IMAGE_SPI_SPEED_DIV_3.
- Enum ESP_IMAGE_SPI_SPEED_20M has been renamed to ESP_IMAGE_SPI_SPEED_DIV_4.

Task Watchdog Timers

- The API for esp_task_wdt_init() has changed as follows:
  - Configuration is now passed as a configuration structure.
  - The function will now handle subscribing of the idle tasks if configured to do so.
- The former CONFIG_ESP_Task_WDT configuration option has been renamed to CONFIG_ESP_TASK_WDT_INIT and a new CONFIG_ESP_TASK_WDT_EN option has been introduced.

FreeRTOS
Legacy API and Data Types  Previously, the `configENABLE_BACKWARD_COMPATIBILITY` option was set by default, thus allowing pre FreeRTOS v8.0.0 function names and data types to be used. The `configENABLE_BACKWARD_COMPATIBILITY` is now disabled by default, thus legacy FreeRTOS names/types are no longer supported by default. Users should do one of the followings:

- Update their code to remove usage of legacy FreeRTOS names/types.
- Enable the `CONFIG_FREERTOS_ENABLE_BACKWARD_COMPATIBILITY` to explicitly allow the usage of legacy names/types.

Tasks Snapshot  The header `task_snapshot.h` has been removed from `freertos/task.h`. ESP-IDF developers should include `freertos/task_snapshot.h` if they need tasks snapshot API.

The function `vTaskGetSnapshot()` now returns `BaseType_t`. Return value shall be `pdTRUE` on success and `pdFALSE` otherwise.

FreeRTOS Asserts  Previously, FreeRTOS asserts were configured separately from the rest of the system using the `FREERTOS_ASSERT` kconfig option. This option has now been removed and the configuration is now done through `COMPILER_OPTIMIZATION_ASSERTION_LEVEL`.

Port Macro API  The file `portmacro_deprecated.h` which was added to maintain backward compatibility for deprecated APIs is removed. Users are advised to use the alternate functions for the deprecated APIs as listed below:

- `portENTER_CRITICAL_NESTED()` is removed. Users should use the `portSET_INTERRUPT_MASK_FROM_ISR()` macro instead.
- `portEXIT_CRITICAL_NESTED()` is removed. Users should use the `portCLEAR_INTERRUPT_MASK_FROM_ISR()` macro instead.
- `vPortCPUInitializeMutex()` is removed. Users should use the `spinlock_initialize()` function instead.
- `vPortCPUAcquireMutex()` is removed. Users should use the `spinlock_acquire()` function instead.
- `vPortCPUAcquireMutexTimeout()` is removed. Users should use the `spinlock_acquire()` function instead.
- `vPortCPUReleaseMutex()` is removed. Users should use the `spinlock_release()` function instead.

App Update

- The functions `esp_ota_get_app_description()` and `esp_ota_get_app_elf_sha256()` have been termed as deprecated. Please use the alternative functions `esp_app_get_description()` and `esp_app_get_elf_sha256()` respectively. These functions have now been moved to a new component `esp_app_format`. Please refer to the header file `esp_app_desc.h`.

Bootloader Support

- The `esp_app_desc_t` structure, which used to be declared in `esp_app_format.h`, is now declared in `esp_app_desc.h`.
- The function `bootloader_common_get_partition_description()` has now been made private. Please use the alternative function `esp_ota_get_partition_description()`. Note that this function takes `esp_partition_t` as its first argument instead of `esp_partition_pos_t`.

Chip Revision  The bootloader checks the chip revision at the beginning of the application loading. The application can only be loaded if the version is `>= CONFIG_ESP32S2_REV_MIN` and `< CONFIG_ESP32S2_REV_MAX_FULL`.  

Espressif Systems 2152  Release master

Submit Document Feedback
During the OTA upgrade, the version requirements and chip revision in the application header are checked for compatibility. The application can only be updated if the version is \( \geq CONFIG\_ESP32S2\_REV\_MIN \) and \( < CONFIG\_ESP32S2\_REV\_MAX\_FULL \).

**Tools**

**ESP-IDF Monitor** ESP-IDF Monitor makes the following changes regarding baud-rate:

- ESP-IDF monitor now uses the custom console baud-rate (\( CONFIG\_ESP\_CONSOLE\_UART\_BAUDRATE \)) by default instead of 115200.
- Setting a custom baud from menuconfig is no longer supported.
- A custom baud-rate can be specified from command line with the `idf.py monitor -b <baud>` command or through setting environment variables.
- Please note that the baud-rate argument has been renamed from `-B` to `-b` in order to be consistent with the global baud-rate `idf.py -b <baud>`. Run `idf.py monitor --help` for more information.

**Deprecated Commands** `idf.py` sub-commands and `cmake` target names have been unified to use hyphens (\(-\)) instead of underscores (\(_\)). Using a deprecated sub-command or target name will produce a warning. Users are advised to migrate to using the new sub-commands and target names. The following changes have been made:

<table>
<thead>
<tr>
<th>Old Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>efuse_common_table</td>
<td>efuse-common-table</td>
</tr>
<tr>
<td>efuse_custom_table</td>
<td>efuse-custom-table</td>
</tr>
<tr>
<td>erase_flash</td>
<td>erase-flash</td>
</tr>
<tr>
<td>partition_table</td>
<td>partition-table</td>
</tr>
<tr>
<td>partition_table-flash</td>
<td>partition-table-flash</td>
</tr>
<tr>
<td>post_debug</td>
<td>post-debug</td>
</tr>
<tr>
<td>show_efuse_table</td>
<td>show-efuse-table</td>
</tr>
<tr>
<td>erase_otadata</td>
<td>erase-otadata</td>
</tr>
<tr>
<td>read_otadata</td>
<td>read-otadata</td>
</tr>
</tbody>
</table>

**Esptool** The `CONFIG\_ESPTOOLPY\_FLASHSIZE\_DETECT` option has been renamed to `CONFIG\_ESPTOOLPY\_HEADER\_FLASHSIZE\_UPDATE` and has been disabled by default. New and existing projects migrated to ESP-IDF v5.0 have to set `CONFIG\_ESPTOOLPY\_FLASHSIZE`. If this is not possible due to an unknown flash size at build time, then `CONFIG\_ESPTOOLPY\_HEADER\_FLASHSIZE\_UPDATE` can be enabled. However, once enabled, to keep the digest valid, an SHA256 digest is no longer appended to the image when updating the binary header with the flash size during flashing.

**Windows Environment** The Msys/Mingw-based Windows environment support got deprecated in ESP-IDF v4.0 and was entirely removed in v5.0. Please use `ESP-IDF Tools Installer` to set up a compatible environment. The options include Windows Command Line, Power Shell and the graphical user interface based on Eclipse IDE. In addition, a VS Code-based environment can be set up with the supported plugin: https://github.com/espressif/vscode-esp-idf-extension.

### 6.1.2 Migration from 5.0 to 5.1
GCC

GCC Version  The previous GCC version was GCC 11.2.0. This has now been upgraded to GCC 12.2.0 on all targets. Users that need to port their code from GCC 11.2.0 to 12.2.0 should refer to the series of official GCC porting guides listed below:

- Porting to GCC 12

Warnings  The upgrade to GCC 12.2.0 has resulted in the addition of new warnings, or enhancements to existing warnings. The full details of all GCC warnings can be found in GCC Warning Options. Users are advised to double-check their code, then fix the warnings if possible. Unfortunately, depending on the warning and the complexity of the user’s code, some warnings will be false positives that require non-trivial fixes. In such cases, users can choose to suppress the warning in multiple ways. This section outlines some common warnings that users are likely to encounter and ways to fix them.

-Wuse-after-free  Typically, this warning should not produce false-positives for release-level code. But this may appear in test cases. There is an example of how it was fixed in ESP-IDF’s `test_realloc.c`.

```c
void *x = malloc(64);
void *y = realloc(x, 48);
TEST_ASSERT_EQUAL_PTR(x, y);
```

Pointers may be converted to int to avoid warning `-Wuse-after-free`.

```c
int x = (int) malloc(64);
int y = (int) realloc((void *) x, 48);
TEST_ASSERT_EQUAL_UINT32((uint32_t) x, (uint32_t) y);
```

-Waddress  GCC 12.2.0 introduces an enhanced version of the `-Waddress` warning option, which is now more eager in detecting the checking of pointers to an array in if-statements.

The following code triggers the warning:

```c
char array[8];
...
if (array)
    memset(array, 0xff, sizeof(array));
```

Eliminating unnecessary checks resolves the warning.

```c
char array[8];
...
memset(array, 0xff, sizeof(array));
```

RISC-V Builds Outside of ESP-IDF  The RISC-V extensions `zicsr` and `zifencei` have been separated from the `I` extension. GCC 12 reflects this change, and as a result, when building for RISC-V ESP32 chips outside of the ESP-IDF framework, you must include the `_zicsr_zifencei` postfix when specifying the `-march` option in your build system.

Example:

```bash
riscv32-esp-elf-gcc main.c -march-rv32imac
```

Now is replaced with:
Peripheral

DAC  DAC driver has been redesigned (see DAC API Reference), which aims to unify the interface and extend the usage of DAC peripheral. Although it is recommended to use the new driver APIs, the legacy driver is still available in the previous include path `driver/dac.h`. However, by default, including `driver/dac.h` will bring a build warning like The legacy DAC driver is deprecated, please use 'driver/dac_oneshot.h', 'driver/dac_cosine.h' or 'driver/dac_continuous.h' instead. The warning can be suppressed by the Kconfig option CONFIG_DAC_SUPPRESS_DEPRECATED_WARN. The major breaking changes in concept and usage are listed as follows:

Breaking Changes in Concepts

- `dac_channel_t` which was used to identify the hardware channel are removed from user space. The channel index now starts from 0, so please use `DAC_CHAN_0` and `DAC_CHAN_1` instead. And in the new driver, DAC channels can be selected by using `dac_channel_mask_t`. And these channels can be allocated in a same channel group which is represented by `dac_channels_handle_t`.
- `dac_cw_scale_t` is replaced by `dac_cosine_atten_t` to decouple the legacy driver and the new driver.
- `dac_cw_phase_t` is replaced by `dac_cosine_phase_t`. The enumerate value is now the phase angle directly.
- `dac_cw_config_t` is replaced by `dac_cosine_config_t`, but the `en_ch` field is removed because it should be specified while allocating the channel group.
- `dac_digi_convert_mode_t` is removed. The driver now can transmit DMA data in different ways by calling `dac_channels_write_continuously()` or `dac_channels_write_cyclically()`.
- `dac_digi_config_t` is replaced by `dac_continuous_config_t`.

Breaking Changes in Usage

- `dac_pad_get_io_num` is removed.
- `dac_output_voltage` is replaced by `dac_oneshot_output_voltage()`.
- `dac_output_enable` is removed. For oneshot mode, it will be enabled after the channel is allocated.
- `dac_output_disable` is removed. For oneshot mode, it will be disabled before the channel is deleted.
- `dac_cw_generator_enable` is replaced by `dac_cosine_start()`.
- `dac_cw_generator_disable` is replaced by `dac_cosine_stop()`.
- `dac_cw_generator_config` is replaced by `dac_cosine_new_channel()`.
- `dac_digi_init` and `dac_digi_controller_config` is merged into `dac_continuous_new_channels()`.
- `dac_digi_deinit` is replaced by `dac_continuous_del_channels()`.
- `dac_digi_start`, `dac_digi_fifo_reset` and `dac_digi_reset` are merged into `dac_continuous_enable()`.
- `dac_digi_stop` is replaced by `dac_continuous_disable()`.

GPSPI  Following items are deprecated. Since ESP-IDF v5.1, GPSPI clock source is configurable.

- `spi_get_actual_clock` is deprecated, you should use `spi_device_get_actual_freq()` instead.
Chapter 6. Migration Guides

LEDC

- `soc_periph_ledc_clk_src_legacy_t::LEDC_USE_RTC8M_CLK` is deprecated. Please use `LEDC_USE_RC_FAST_CLK` instead.

Storage

**FatFs** `esp_vfs_fat_sdmmc_unmount()` is now deprecated, and you can use `esp_vfs_fat_sdcard_unmount()` instead. This API is deprecated in previous ESP-IDF versions, but without a deprecation warning or migration guide. Since ESP-IDF v5.1, calling this `esp_vfs_fat_sdmmc_unmount()` API will generate a deprecation warning.

**SPI FLASH**

- `spi_flash_get_counters()` is deprecated, please use `esp_flash_get_counters()` instead.
- `spi_flash_dump_counters()` is deprecated, please use `esp_flash_dump_counters()` instead.
- `spi_flash_reset_counters()` is deprecated, please use `esp_flash_reset_counters()` instead.

Networking

**SNTP** SNTP module now provides thread safe APIs to access lwIP functionality. It is recommended to use `ESP_NETIF` API. Please refer to the chapter `SNTP API` for more details.

System

FreeRTOS

Dynamic Memory Allocation

In the past, FreeRTOS commonly utilized the function `malloc()` to allocate dynamic memory. As a result, if an application allowed `malloc()` to allocate memory from external RAM (by configuring the `CONFIG_SPIRAM_USE` option as `CONFIG_SPIRAM_USE_MALLOC`), FreeRTOS had the potential to allocate dynamic memory from external RAM, and the specific location was determined by the heap allocator.

**Note:** Dynamic memory allocation for tasks (which are likely to consume the most memory) were an exception to the scenario above. FreeRTOS would use a separate memory allocation function to guarantee that dynamic memory allocate for a task was always placed in internal RAM.

Allowing FreeRTOS objects (such as queues and semaphores) to be placed in external RAM becomes an issue if those objects are accessed while the cache is disabled (such as during SPI flash write operations) and would lead to a cache access errors (see `Fatal Errors` for more details).
Therefore, FreeRTOS has been updated to always use internal memory (i.e., DRAM) for dynamic memory allocation. Calling FreeRTOS creation functions (e.g., `xTaskCreate()`, `xQueueCreate()`) guarantees that the memory allocated for those tasks/objects is from internal memory (see FreeRTOS Heap for more details).

**Warning:** If you previously relied on `CONFIG_SPIRAM_USE` to place FreeRTOS objects into external memory, this change will lead to increased usage of internal memory due the FreeRTOS objects now being allocated there.

To place a FreeRTOS task/object into external memory, it is now necessary to do so explicitly. The following methods can be employed:

- Allocate the task/object using one of the ...CreateWithCaps() API such as `xTaskCreateWithCaps()` and `xQueueCreateWithCaps()` (see IDF Additional API for more details).
- Manually allocate external memory for those objects using `heap_caps_malloc()`, then create the objects from the allocated memory using one of the ...CreateStatic() FreeRTOS functions.

**Power Management**

- `esp_pm_config_esp32xx_t` is deprecated, use `esp_pm_config_t` instead.
- `esp32xx/pm.h` is deprecated, use `esp_pm.h` instead.

### 6.1.3 Migration from 5.1 to 5.2

**GCC**

**GCC Version** The previous GCC version was GCC 12.2.0. This has now been upgraded to GCC 13.2.0 on all targets. Users that need to port their code from GCC 12.2.0 to 13.2.0 should refer to the series of official GCC porting guides listed below:

- Porting to GCC 13

**Common Porting Problems and Fixes**

**stdio.h No Longer Includes `sys/types.h`**

**Issue** Compilation errors may occur in code that previously worked with the old toolchain. For example:

```c
#include <stdio.h>
clock_t var; // error: expected specifier-qualifier-list before 'clock_t'
```

**Solution** To resolve this issue, the correct header must be included. Refactor the code like this:

```c
#include <time.h>
clock_t var;
```

**Peripherals**
UART

• UART_FIFO_LEN is deprecated. Please use UART_HW_FIFO_LEN instead.

I2C

I2C driver has been redesigned (see I2C API Reference), which aims to unify the interface and extend the usage of I2C peripheral. Although it is recommended to use the new driver APIs, the legacy driver is still available in the previous include path driver/i2c.h.

The major breaking changes in concept and usage are listed as follows:

Major Changes in Concepts

• i2c_config_t which was used to configure the I2C bus, but it doesn’t really tell whether to configure master or slave. So in the new design, master and slave initialization are separate, user can call i2c_master_bus_config_t or i2c_slave_config_t.
• i2c_mode_t which was used to tell whether I2C controller works in slave mode or master mode. This enumerator has been deprecated. In the new driver, users don’t need to manually set the mode anymore since master and slave APIs are different.
• i2c_rw_t which was used to tell whether I2C master controller is performing a write or a read operation. This is now deprecated.
• i2c_addr_mode_t was renamed to i2c_addr_bit_len_t.
• In the legacy driver, operations needed to be chained with a command list (dynamically or statically created). The new driver now handles this internally, making the operations more size and space efficient.
• Capability flags like I2C_SCLK_SRC_FLAG_FOR_NOMAL are used to select clock source in the legacy driver. In the new driver, users can select clock source directly.

Major Changes in Usage

• I2C bus initialization is done in two parts: first, initialization of the bus with i2c_new_master_bus(), then, initialization of the I2C device with i2c_master_bus_add_device().
• i2c_reset_tx_fifo and i2c_reset_rx_fifo have been removed, since it is never required to reset the fifo by users. Whole bus can still be reset by calling i2c_master_bus_reset.
• i2c_cmd_link_xxx functions have been removed, user doesn’t need to use link to link commands on its own.
• i2c_master_write_to_device has been renamed to i2c_master_transmit.
• i2c_master_read_from_device has been renamed to i2c_master_receive.
• i2c_master_write_read_device has been renamed to i2c_master_transmit_receive.
• i2c_slave_write_buffer has been renamed to i2c_slave_transmit.
• i2c_slave_read_buffer has been renamed to i2c_slave_receive.

Protocols

CoAP

CoAP examples have been moved to idf-extra-components repository.

HTTP2

http2_request example has been moved to idf-extra-components repository.

System

FreeRTOS
Chapter 6. Migration Guides

**Task Snapshot**  The Task Snapshot API has been made private due to a lack of a practical way for the API to be used from user code (the scheduler must be halted before the API can be called).

As a result, the `#include "freertos/task_snapshot.h"` include path has been deprecated.

**WiFi**

**WiFi Enterprise Security**  APIs defined in `esp_wpa2.h` have been deprecated. Please use newer APIs from `esp_eap_client.h`.

### 6.1.4 Migration from 5.2 to 5.3

**Peripherals**

In order to control the dependence of other components on drivers at a smaller granularity, the original peripheral drivers under the `driver` component were split into separate components:

- `esp_driver_gptimer` - Driver for general purpose timers
- `esp_driver_pcnt` - Driver for pulse counter
- `esp_driver_gpio` - Driver for GPIO
- `esp_driver_spi` - Driver for GPSPI
- `esp_driver_mcpwm` - Driver for Motor Control PWM
- `esp_driver_sdmmc` - Driver for SDMMC

For compatibility, the original `driver` component is still treated as an all-in-one component by registering these `esp_driver_xyz` components as its public dependencies. In other words, you do not need to modify the CMake file of an existing project, but you now have a way to specify the specific peripheral driver that your project depends on.

Originally, you may have used `linker.lf` to specify the link location of some driver functions in memory space, but now, because the location of the driver files have been moved, you need to make changes your `linker.lf` file accordingly. For example, a linker.lf file with the following entries:

```
[mapping:my_mapping_scheme]
archive: libdriver.a
entries:
  gpio (noflash)
```

Should be changed to:

```
[mapping:my_mapping_scheme]
archive: libesp_driver_gpio.a
entries:
  gpio (noflash)
```
Chapter 7

Libraries and Frameworks

7.1 Cloud Frameworks

ESP32-S2 supports multiple cloud frameworks using agents built on top of ESP-IDF. Here are the pointers to various supported cloud frameworks’ agents and examples:

7.1.1 ESP RainMaker

ESP RainMaker is a complete solution for accelerated AIoT development. ESP RainMaker on GitHub.

7.1.2 AWS IoT

https://github.com/espressif/esp-aws-iot is an open source repository for ESP32-S2 based on Amazon Web Services’ aws-iot-device-sdk-embedded-C.

7.1.3 Azure IoT

https://github.com/espressif/esp-azure is an open source repository for ESP32-S2 based on Microsoft Azure’s azure-iot-sdk-c SDK.

7.1.4 Google IoT Core

https://github.com/espressif/esp-google-iot is an open source repository for ESP32-S2 based on Google’s iot-device-sdk-embedded-c SDK.

7.1.5 Aliyun IoT

https://github.com/espressif/esp-aliyun is an open source repository for ESP32-S2 based on Aliyun’s iotkit-embedded SDK.

7.1.6 Joylink IoT

https://github.com/espressif/esp-joylink is an open source repository for ESP32-S2 based on Joylink’s joylink_dev_sdk SDK.
Chapter 7. Libraries and Frameworks

7.1.7 Tencent IoT

https://github.com/espressif/esp-welink is an open source repository for ESP32-S2 based on Tencent’s welink SDK.

7.1.8 Tencentyun IoT

https://github.com/espressif/esp-qcloud is an open source repository for ESP32-S2 based on Tencentyun’s qcloud-iot-sdk-embedded-c SDK.

7.1.9 Baidu IoT

https://github.com/espressif/esp-baidu-iot is an open source repository for ESP32-S2 based on Baidu’s iot-sdk-c SDK.

7.2 Espressif’s Frameworks

Here you will find a collection of the official Espressif libraries and frameworks.

7.2.1 Espressif Audio Development Framework

The ESP-ADF is a comprehensive framework for audio applications including:

- CODEC’s HAL
- Music players and recorders
- Audio processing
- Bluetooth speakers
- Internet radios
- Hands-free devices
- Speech decognition

This framework is available on GitHub: ESP-ADF.

7.2.2 ESP-CSI

ESP-CSI is an experimental implementation that uses the Wi-Fi Channel State Information to detect the presence of a human body.

See the ESP-CSI project for more information.

7.2.3 Espressif DSP Library

The library provides algorithms optimized specifically for digital signal processing applications. This library supports:

- Matrix multiplication
- Dot product
- FFT (Fast Fourier Transform)
- IIR (Infinite Impulse Response)
- FIR (Finite Impulse Response)
- Vector math operations

This library is available on Github: ESP-DSP library.
7.2.4 ESP-WIFI-MESH Development Framework

This framework is based on the ESP-WIFI-MESH protocol with the following features:

- Fast network configuration
- Stable upgrade
- Efficient debugging
- LAN control
- Various application demos

This framework is available on Github: ESP-MDF.

7.2.5 ESP-WHO

The ESP-WHO is a face detection and recognition framework using the ESP32 and camera.

This framework is available on Github: ESP-WHO.

7.2.6 ESP RainMaker

ESP RainMaker is a complete solution for accelerated AIoT development. Using ESP RainMaker, you can create AIoT devices from the firmware to the integration with voice-assistant, phone apps and cloud backend.

This project is available on Github: ESP RainMaker on GitHub.

7.2.7 ESP-IoT-Solution

ESP-IoT-Solution contains commonly used device drivers and code frameworks when developing IoT systems. The device drivers and code frameworks within the ESP-IoT-Solution are organized as separate components, allowing them to be easily integrated into an ESP-IDF project.

ESP-IoT-Solution includes:

- Device drivers for sensors, display, audio, GUI, input, actuators, etc.
- Framework and documentation for low power, security, storage, etc.
- Guide for Espressif open source solutions from practical application point.

This solution is available on Github: ESP-IoT-Solution on GitHub.

7.2.8 ESP-Protocols

The ESP-Protocols repository contains a collection of protocol components for ESP-IDF. The code within ESP-Protocols is organized into separate components, allowing them to be easily integrated into an ESP-IDF project. Additionally, each component is available in IDF Component Registry.

ESP-Protocols components:

- esp_modem enables connectivity with GSM/LTE modems using AT commands or PPP protocol. See the esp_modem documentation.
- mdns (mDNS) is a multicast UDP service that is used to provide local network service and host discovery. See the mdns documentation.
- esp_websocket_client is a managed component for ESP-IDF that contains implementation of WebSocket protocol client for ESP32. See the esp_websocket_client documentation. For details of WebSocket protocol client, see WebSocket_protocol_client.
- asio is a cross-platform C++ library, see https://think-async.com/Asio/. It provides a consistent asynchronous model using a modern C++ approach. See the asio documentation.
7.2.9 ESP-BSP

The ESP-BSP repository contains Board Support Packages (BSPs) for various Espressif’s and third-party development boards. BSPs help to quickly get started with a supported board. Usually they contain pinout definition and helper functions that will initialize peripherals for the specific board. Additionally, the BSPs contain drivers for external chips populated on the development board, such as sensors, displays, audio codecs, etc.

7.2.10 ESP-IDF-CXX

ESP-IDF-CXX contains C++ wrappers for part of ESP-IDF. The focuses are on ease of use, safety, automatic resource management. They also move error checking from runtime to compile time to prevent running failure. There are C++ classes for ESP-Timer, I2C, SPI, GPIO and other peripherals or features of ESP-IDF. ESP-IDF-CXX is available as a component from the component registry. Please check the project’s README.md for more information.
Chapter 8

Contributions Guide

We welcome contributions to the ESP-IDF project!

8.1 How to Contribute

Contributions to ESP-IDF - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via Github Pull Requests.

8.2 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. Please check the Copyright Header Guide for additional information.
- Does any new code conform to the ESP-IDF Style Guide?
- Have you installed the pre-commit hook for ESP-IDF project?
- 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 are unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

8.3 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 into the public GitHub repository.
Chapter 8. Contributions Guide

8.4 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.

8.5 Related Documents

8.5.1 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

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, i.e. 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 (i.e. shortening data to dat), unless the resulting name would otherwise be very long.

Indentation

Use 4 spaces for each indentation level. Do not 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. Do not begin or end a function with an empty line.

```c
void function1()
{
    do_one_thing();
    do_another_thing();
    // INCORRECT, do not place empty line here
}

void function2()
{
    // INCORRECT, do not use an empty line here
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
        // place empty line here
    }
}
```

(continues on next page)
The maximum line length is 120 characters as long as it does not seriously affect the readability.

**Horizontal Space** Always add single space after conditional and loop keywords:

```c
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:

```c
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;
++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:

```c
esp_rom_gpio_connect_in_signal(PIN_CAM_D6,   I2S0I_DATA_IN15_IDX,  false);
esp_rom_gpio_connect_in_signal(PIN_CAM_D7,   I2S0I_DATA_IN15_IDX,  false);
esp_rom_gpio_connect_in_signal(PIN_CAM_HREF, I2S0I_H_ENABLE_IDX,   false);
esp_rom_gpio_connect_in_signal(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:

```c
// This is correct:
void function(int arg)
{
}
```

(continues on next page)
• Within a function, place opening brace on the same line with conditional and loop statements:

```c
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.

• Do not use single comments to disable some functionality:

```c
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:

```c
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. Do not use `#if 0 ... #endif` or comments to store code snippets which you may need in the future.

• Do not 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:

```c
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>.
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 --no-generic options dos2unix && git commit -a --amend --no-edit --allow-empty'
```

(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 is possible to run `dos2unix FILENAME` and then run `git commit --amend`.

**Formatting Your Code**  ESP-IDF uses Astyle to format source code. The configuration is stored in `tools/ci/astyle-rules.yml` file.

Initially, all components are excluded from formatting checks. You can enable formatting checks for the component by removing it from `components_not_formatted_temporary` list. Then run:

```
pre-commit run --files <path_to_files> astyle_py
```

Alternatively, you can run `astyle_py` manually. You can install it with `pip install astyle_py==VERSION`. Make sure you have the same version installed as the one specified in `.pre-commit-config.yaml` file. With `astyle_py` installed, run:

```
astyle_py --rules=$IDF_PATH/tools/ci/astyle-rules.yml <path-to-file>
```

**Type Definitions**  Should be snake_case, ending with _t suffix:

```
typedef int signed_32_bit_t;
```

**Enum**  Enums should be defined through the `typedef` and be namespaced:

```
typedef enum
{
    MODULE_FOO_ONE,
    MODULE_FOO_TWO,
    MODULE_FOO_THREE
} module_foo_t;
```

**Assertions**  The standard C `assert()` function, defined in `assert.h` should be used to check conditions that should be true in source code. In the default configuration, an assert condition that returns `false` or `0` will call `abort()` and trigger a **Fatal Error**.

`assert()` should only be used to detect unrecoverable errors due to a serious internal logic bug or corruption, where it is not possible for the program to continue. For recoverable errors, including errors that are possible due to invalid external input, an **error value should be returned**.

**Note:**  When asserting a value of type `esp_err_t` is equal to `ESP_OK`, use the `ESP_ERROR_CHECK Macro` instead of an `assert()`.

It is possible to configure ESP-IDF projects with assertions disabled (see `CONFIG_COMPILER_OPTIMIZATION_ASSERTION_LEVEL`). Therefore, functions called in an `assert()` statement should not have side-effects.

It is also necessary to use particular techniques to avoid “variable set but not used” warnings when assertions are disabled, due to code patterns such as:
int res = do_something();
assert(res == 0);

Once the assert is optimized out, the res value is unused and the compiler will warn about this. However the function do_something() must still be called, even if assertions are disabled.

When the variable is declared and initialized in a single statement, a good strategy is to cast it to void on a new line. The compiler will not produce a warning, and the variable can still be optimized out of the final binary:

int res = do_something();
assert(res == 0);
(void *)res;

If the variable is declared separately, for example if it is used for multiple assertions, then it can be declared with the GCC attribute __attribute__((unused)). The compiler will not produce any unused variable warnings, but the variable can still be optimized out:

int res __attribute__((unused));
res = do_something();
assert(res == 0);
res = do_something_else();
assert(res != 0);

Header File Guards

All public facing header files should have preprocessor guards. A pragma is preferred:

#pragma once

over the following pattern:

#ifndef FILE_NAME_H
#define FILE_NAME_H
...
#endif // FILE_NAME_H

In addition to guard macros, all C header files should have extern "C" guards to allow the header to be used from C++ code. Note that the following order should be used: pragma once, then any #include statements, then extern "C" guards:

#pragma once
#include <stdint.h>
#ifndef __cplusplus
extern "Cplusplus extern "C" {
#endif /* declarations go here */
#endif __cplusplus
}
#endif

Include Statements

When writing #include statements, try to maintain the following order:
Chapter 8. Contributions Guide

- C standard library headers.
- Other POSIX standard headers and common extensions to them (such as sys/queue.h.)
- Common IDF headers (esp_log.h, esp_system.h, esp_timer.h, esp_sleep.h, etc.)
- Headers of other components, such as FreeRTOS.
- Public headers of the current component.
- Private headers.

Use angle brackets for C standard library headers and other POSIX headers (#include <stdio.h>).

Use double quotes for all other headers (#include "esp_log.h").

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 .hpp. C++ source files have the extension .cpp. The latter 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. An exception from CamelCase is if the readability is severely decreased, e.g., in GPIOOutput, then an underscore _ is allowed to make it more readable: GPIO_Output.
- Namespaces shall be in lower snake_case.
- Templates are specified in the line above the function declaration.
- Interfaces in terms of Object-Oriented Programming shall be named without the suffix ...Interface. Later, this makes it easier to extract interfaces from normal classes and vice versa without making a breaking change.

Member Order in Classes  In order of precedence:

- First put the public members, then the protected, then private ones. Omit public, protected or private sections without any members.
- First put constructors/destructors, then member functions, then member variables.

For example:

```cpp
class ForExample {
public:
    // first constructors, then default constructor, then destructor
    ForExample(double example_factor_arg);
    ForExample();
    ~ForExample();

    // then remaining public methods
    set_example_factor(double example_factor_arg);

    // then public member variables
    uint32_t public_data_member;

private:
    // first private methods
    void internal_method();

    // then private member variables
    double example_factor;
};
```
Spacing

- Do not indent inside namespaces.
- Put public, protected and private labels at the same indentation level as the corresponding class label.

Simple Example

```c++
// 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 { 
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;
}

(continues on next page)```
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).
- Do not 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.

Third Party Component Code Styles

ESP-IDF integrates a number of third party components where these components may have differing code styles.

FreeRTOS The code style adopted by FreeRTOS is described in the FreeRTOS style guide. Formatting of FreeRTOS source code is automated using Uncrustify, thus a copy of the FreeRTOS code style’s Uncrustify configuration (uncrustify.cfg) is stored within ESP-IDF FreeRTOS component.

If a FreeRTOS source file is modified, the updated file can be formatted again by following the steps below:

1. Ensure that Uncrustify (v0.69.0) is installed on your system
2. Run the following command on the update FreeRTOS source file (where source.c is the path to the source file that requires formatting).

```
uncrustify -c $IDF_PATH/components/freertos/FreeRTOS-Kernel/uncrustify.cfg --
←replace source.c --no-backup
```

Documenting Code

Please see the guide here: Documenting Code.
8.5.2 Install Pre-commit Hook for ESP-IDF Project

Required Dependency

Python 3.8.* or above. This is our recommended Python version for ESP-IDF developers.
If you still have Python versions not compatible, update your Python versions before installing the pre-commit hook.

Install pre-commit

Run `pip install pre-commit`

Install pre-commit Hook

1. Go to the ESP-IDF project directory.
2. Run `pre-commit install --allow-missing-config -t pre-commit -t commit-msg`. Install hook by this approach will let you commit successfully even in branches without the `.pre-commit-config.yaml`
3. pre-commit hook will run automatically when you are running `git commit` command

Uninstall pre-commit Hook

Run `pre-commit uninstall`

What Is More?

For detailed usage, please refer to the documentation of `pre-commit`.

Common Problems For Windows Users

`/usr/bin/env: python: Permission denied.`

If you are in Git Bash, please check the python executable location by run `which python`.

If the executable is under `~/AppData/Local/Microsoft/WindowsApps/`, then it is a link to Windows AppStore, not a real one.

Please install Python manually and update this in your PATH environment variable.

Your USERPROFILE contains non-ASCII characters

`pre-commit` may fail when initializing an environment for a particular hook when the path of `pre-commit`'s cache contains non-ASCII characters. The solution is to set `PRE_COMMIT_HOME` to a path containing only standard characters before running pre-commit.
8.5.3 Documenting Code

The purpose of this description is to provide a quick summary of the 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 \text{is} oxygen to air ratio
*/```

Doxygen can phrase the code, extract the commands together with subsequent text, and build documentation out of it.

Typical comment block, that contains documentation of a function, looks like below:

Doxygen supports a couple of formatting styles. It also gives you great flexibility on the 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 Doxygen?

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, including functions, structs, typedefs, enums, macros, etc. Provide enough information about 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.

4. If function has void input or does not return any value, then skip @param or @return.
5. When documenting a `define` as well as members of a `struct` or `enum`, place specific comment like below after each member.

```
typedef enum {
    NVS_READONLY, /* Read only */
    NVS_READWRITE, /* Read and write */
} nvs_open_mode;
```

```
/* !< how to documented members */
```

6. To provide well-formatted lists, break the line after command (like `@return` in the example below).

```
   * @return
   *     - ESP_OK if erase operation was successful
   *     - ESP_ERR_NVS_INVALID_HANDLE if handle has been closed or is NULL
   *     - ESP_ERR_NVS_READONLY if handle was opened as read only
   *     - ESP_ERR_NVS_NOT_FOUND if the requested key does not 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 a separate README.rst file of the same directory. If this directory contains header files for different APIs, then the file name should be apiname-readme.rst.

Go One Extra Mile

Here are a couple of tips on how you can make your documentation even better and more useful to the reader and writer.

When writing code, please follow the guidelines below:

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.
```

(continues on next page)
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 repetitive defines or enumerations. In such case, enclose the code within `/** @cond */` and `/** @endcond */` commands. Example of such implementation is provided in `esp_driver_gpio/include/driver/gpio.h`.

5. Use markdown to make your documentation even more readable. You will add headers, links, tables and more.

```
```

**Note:** Code snippets, notes, links, etc., will not make it to the documentation, if not enclosed in a comment block associated with one of the documented objects.

6. Prepare one or more complete code examples together with description. Place description to a separate file `README.md` in specific folder of `examples` directory.

**Standardize Document Format**

When it comes to text, please follow guidelines below to provide well-formatted Markdown (.md) or reST (.rst) documents.

1. Please ensure that one paragraph is written in one line. Do not break lines like below. Breaking lines to enhance readability is only suitable for writing code. To make the text easier to read, it is recommended to place an empty line to separate the paragraph.

2. Please make the line number of CN and EN documents consistent like below. The benefit of this approach is that it can save time for both writers and translators. When non-bilingual writers need to update text, they only need to update the same line in the corresponding CN or EN document. For translators, if documents are updated in English, then translators can quickly locate where to update in the corresponding CN document later. Besides, by comparing the total number of lines in EN and CN documents, you can quickly find out whether the CN version lags behind the EN version.
Fig. 1: One line for one paragraph (click to enlarge)

Fig. 2: No line breaks within the same paragraph (click to enlarge)

Fig. 3: Keep the line number for EN and CN documents consistent (click to enlarge)
Chapter 8. Contributions Guide

Building Documentation

The documentation is built with the *esp-docs* Python package, which is a wrapper around *Sphinx*.

To install it simply do:

```
pip install esp-docs
```

After a successful install then the documentation can be built from the docs folder with:

```
build-docs build
```

or for specific target and language with:

```
build-docs -t esp32 -l en build
```

For more in-depth documentation about *esp-docs* features please see the documentation at *esp-docs*.

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*

8.5.4 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 are 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 a pytest_<example name>.py file for running an automated example test. If submitting a GitHub Pull Request which includes an example, it is OK not to include this file initially. The details can be discussed as part of the Pull Request. Please refer to *IDF Tests with Pytest Guide* for details.

General Guidelines

Example code should follow the *Espressif IoT Development Framework Style Guide*. 
Checklist

Checklist before submitting a new example:

- Example project name (in 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.

8.5.5 API Documentation Template

Note: INSTRUCTIONS

1. Use this file (docs/en/api-reference/template.rst) as a template to document API.
2. Change the file name to the name of the header file that represents the 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. Include code snippets to illustrate functionality of particular functions when applicable.
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, and add README.md file.
4. Provide overview of demonstrated functionality in README.md.
5. With good overview readers 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. ESP-IDF repository provides 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 Sphinx extension esp_extensions/run_doxygen.py for all header files listed in the INPUT statement of docs/doxygen/Doxyfile.
3. Each line of the INPUT statement (other than a comment that begins with ##) contains a path to header file *.h that is used to generate corresponding *.inc files:

```
## Wi-Fi - API Reference
##
../components/esp32/include/esp_wifi.h \
../components/esp32/include/esp_smartconfig.h \
```

4. When the headers are expanded, any macros defined by default in sdkconfig.h as well as any macros defined in SOC-specific include/soc/_caps.h headers will be expanded. This allows the headers to include or exclude material based on the IDF_TARGET value.
5. 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, e.g., esp_wifi.h, run python gen-dxd.py esp32/include/esp_wifi.h.
6. To show contents of *.inc file in documentation, include it as follows:

```
.. include-build-file:: inc/esp_wifi.inc
```

For example see docs/en/api-reference/network/esp_wifi.rst


Below is the list of common .. doxygen directives:

- Functions - doxygenfunction:: name_of_function
- Unions - doxygenuinon:: name_of_union
- Structures - doxygenstruct:: name_of_structure together with :members:
- Macros - doxygendifine:: 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 directive as follows:

```
* :component_file:'path_to/header_file.h'
```
8. In any case, to generate API reference, the file `docs/doxygen/Doxyfile` should be updated with paths to `*.h` headers that are being documented.

9. When changes are committed and documentation is built, check how this section has been rendered. Correct annotations in respective header files, if required.

### 8.5.6 Contributor Agreement

**Individual Contributor Non-Exclusive License Agreement Including the Traditional Patent License OPTION**

Thank you for your interest in contributing to this Espressif project hosted on GitHub ("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 in the Contributions Guide.

#### 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 must have had Your employer approve this Agreement or sign the Entity version of this Agreement.

**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 by submitting a comment on GitHub.

**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. COPYRIGHT LICENSE

**2.1 Grant of 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. PATENT LICENSE 3.1 Grant of Patent 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 (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 EXPRESSED 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.
8.5.7 Copyright Header Guide

ESP-IDF is released under the Apache License 2.0 with some additional third-party copyrighted code released under various licenses. For further information please refer to the list of copyrights and licenses.

This page explains how the source code should be properly marked with a copyright header. ESP-IDF uses the Software Package Data Exchange (SPDX) format which is short and can be easily read by humans or processed by automated tools for copyright checks.

How to Check the Copyright Headers

Please make sure you have installed the pre-commit hooks which contain a copyright header checker as well. The checker can suggest a header if it is not able to detect a properly formatted SPDX header.

What If the Checker’s Suggestion Is Incorrect?

No automated checker (no matter how good is) can replace humans. So the developer’s responsibility is to modify the offered header to be in line with the law and the license restrictions of the original code on which the work is based on. Certain licenses are not compatible between each other. Such corner cases will be covered by the following examples.

The checker can be configured with the tools/ci/check_copyright_config.yaml configuration file. Please check the options it offers and consider updating it in order to match the headers correctly.

Common Examples of Copyright Headers

The simplest case is when the code is not based on any licensed previous work, e.g., it was written completely from scratch. Such code can be decorated with the following copyright header and put under the license of ESP-IDF:

```c
/*
 * SPDX-FileCopyrightText: 2015-2023 Espressif Systems (Shanghai) CO LTD
 * *
 * SPDX-License-Identifier: Apache-2.0
 */
```
Less Restrictive Parts of ESP-IDF  

Some parts of ESP-IDF are deliberately under less restrictive licenses in order to ease their re-use in commercial closed source projects. This is the case for ESP-IDF examples which are in Public domain or under the Creative Commons Zero Universal (CC0) license. The following header can be used in such source files:

```
/*
 * SPDX-FileCopyrightText: 2015-2023 Espressif Systems (Shanghai) CO LTD
 * SPDX-License-Identifier: Unlicense OR CC0-1.0
 */
```

The option allowing multiple licenses joined with the OR keyword from the above example can be achieved with the definition of multiple allowed licenses in the `tools/ci/check_copyright_config.yaml` configuration file. Please use this option with care and only selectively for a limited part of ESP-IDF.

Third Party Licenses  

Code licensed under different licenses, modified by Espressif Systems and included in ESP-IDF cannot be licensed under Apache License 2.0 even if the checker suggests it. It is advised to keep the original copyright header and add an SPDX before it.

The following example is a suitable header for a code licensed under the “GNU General Public License v2.0 or later” held by John Doe with some additional modifications done by Espressif Systems:

```
/*
 * SPDX-FileCopyrightText: 1991 John Doe
 * SPDX-License-Identifier: GPL-2.0-or-later
 * SPDX-FileContributor: 2019-2023 Espressif Systems (Shanghai) CO LTD
 */
```

The licenses can be identified and the short SPDX identifiers can be found in the official SPDX license list. Other very common licenses are the GPL-2.0-only, the BSD-3-Clause, and the BSD-2-Clause.

In exceptional case, when a license is not present on the SPDX license list, it can be expressed by using the LicenseRef-[idString] custom license identifier, for example LicenseRef-Special-License. The full license text must be added into the LICENSES directory under Special-License filename.

```
/*
 * SPDX-FileCopyrightText: 2015-2023 Espressif Systems (Shanghai) CO LTD
 * SPDX-License-Identifier: LicenseRef-Special-License
 */
```

Dedicated LicenseRef-Included custom license identifier can be used to express a situation when the custom license is included directly in the source file.

```
/*
 * SPDX-FileCopyrightText: 2015-2023 Espressif Systems (Shanghai) CO LTD
 * SPDX-License-Identifier: LicenseRef-Included
 * <Full custom license text>
 */
```

The configuration stored in `tools/ci/check_copyright_config.yaml` offers features useful for third party licenses:

- A different license can be defined for the files part of a third party library.
- The check for a selected set of files can be permanently disabled. Please use this option with care and only in cases when none of the other options are suitable.
8.5.8 pytest in ESP-IDF

ESP-IDF has numerous types of tests that are meant to be executed on an ESP chip (known as on target testing). Target tests are usually compiled as part of an IDF project used for testing (known as a test app), where test apps follow the same build, flash, and monitor process of any other standard IDF project.

Typically, on target testing will require a connected host (e.g., a PC) that is responsible for triggering a particular test case, providing test data, and inspecting test results.

ESP-IDF uses the pytest framework (and some pytest plugins) on the host side to automate on target testing. This guide introduces pytest in ESP-IDF and covers the following concepts:

1. The different types of test apps in ESP-IDF.
2. Using the pytest framework in Python scripts to automate target testing.
3. ESP-IDF Continuous Integration (CI) target testing process.
4. How to run target tests locally with pytest.
5. pytest tips and tricks.

Note: In ESP-IDF, we use the following pytest plugins by default:

- pytest-embedded with default services esp, idf
- pytest-rerunfailures

All the concepts and usages introduced in this guide are based on the default behavior of these plugins, thus may not be available in vanilla pytest.

Installation

All dependencies could be installed by running the install script with the --enable-pytest argument:

```
$ install.sh --enable-pytest
```

Common Issues During Installation

No Package ‘dbus-1’ Found

```bash
configure: error: Package requirements (dbus-1 >= 1.8) were not met:
No package 'dbus-1' found
Consider adjusting the PKG_CONFIG_PATH environment variable if you installed software in a non-standard prefix.
```

If you encounter the error message above, you may need to install some missing packages.

If you are using Ubuntu, you may need to run:

```
sudo apt-get install libdbus-glib-1-dev
```

or

```
sudo apt-get install libdbus-1-dev
```

For other Linux distributions, please Google the error message above and find which missing packages need to be installed for your particular distribution.
**Invalid Command**  ‘`bdist_wheel’`  

error: invalid command 'bdist_wheel'

If you encounter the error message above, you may need to install some missing Python packages such as:

```
python -m pip install -U pip
```

or

```
python -m pip install wheel
```

---

**Note:** Before running the pip commands, please make sure you are using the IDF Python virtual environment.

---

**Test Apps**

ESP-IDF contains different types of test apps that can be automated using pytest.

**Component Tests**  ESP-IDF components typically contain component specific test apps that execute component specific unit tests. Component test apps are the recommended way to test components. All the test apps should be located under `${IDF_PATH}/components/<COMPONENT_NAME>/test_apps`, for example:

```
components/
  └── my_component/
      ├── include/
      │   └── ...
      ├── test_apps/
      │   └── test_app_1
      │       ├── main/
      │       │   └── ...
      │       ├── CMakeLists.txt
      │       │   └── pytest_my_component_app_1.py
      │       └── test_app_2
      │           └── pytest_my_component_app_2.py
      │   └── parent_folder
      │       └── test_app_3
      │           └── pytest_my_component_app_3.py
      │   └── ...
      └── my_component.c
          └── CMakeLists.txt
```

**Example Tests**  The purpose of ESP-IDF examples is to demonstrate parts of ESP-IDF functionality to users (refer to [Examples Readme](#) for more information).

However, to ensure that these examples operate correctly, examples can be treated as test apps and executed automatically by using pytest. All examples should be located under `${IDF_PATH}/examples`, with tested example including a Python test script, for example:

```
examples/
  └── parent_folder/
      └── example_1/
          └── main/
              └── ...
          └── CMakeLists.txt
              └── pytest_example_1.py
```
**Custom Tests** Custom Tests are tests that aim to test some arbitrary functionality of ESP-IDF, thus are not intended to demonstrate IDF functionality to users in any way.

All custom test apps are located under `${IDF_PATH}/tools/test_apps`. For more information please refer to the Custom Test Readme.

**pytest in ESP-IDF**

**pytest Execution Process**

1. **Bootstrapping Phase**

   Create session-scoped caches:
   - port-target cache
   - port-app cache

2. **Collection Phase**

   A. Gather all Python files with the prefix `pytest_`.
   B. Gather all test functions with the prefix `test_`.
   C. Apply the `params`, and duplicate the test functions.
   D. Filter the test cases with CLI options. For the detailed usages, see *Filter the Test Cases*.

3. **Execution Phase**

   A. Construct the **fixtures**. In ESP-IDF, the common fixtures are initialized in this order:
      a. `pexpect_proc`: `pexpect` instance
      b. `app`: `IdfApp` instance
         The test app’s information (e.g., sdkconfig, flash_files, partition_table, etc) would be parsed at this phase.
      c. `serial`: `IdfSerial` instance
         The port of the host to which the target is connected is auto-detected. In the case of multiple targets connected to the host, the test target’s type is parsed from the app. The test app binary files are flashed to the test target automatically.
      d. `dut`: `IdfDut` instance

   B. Run the real test function.

   C. Deconstruct the fixtures in this order:
      a. `dut`
         i. close the `serial` port.
         ii. (Only for apps with **Unity test framework**) generate JUnit report of the Unity test cases.
      b. `serial`
      c. `app`
      d. `pexpect_proc`: Close the file descriptor

   D. (Only for apps with **Unity test framework**) If `dut.expect_from_unity_output()` is called, an `AssertionError` is raised upon detection of a Unity test failure.

4. **Reporting Phase**

   A. Generate JUnit report of the test functions.
   B. Modify the JUnit report test case name into ESP-IDF test case ID format: `<target>.<config>.<test function name>`.

5. **Finalization Phase** (Only for apps with **Unity test framework**) Combine the JUnit reports if the JUnit reports of the Unity test cases are generated.
**Basic Example**  This following Python test script example is taken from pytest_console_basic.py.

```python
@ pytest.mark.esp32
@ pytest.mark.esp32c3
@ pytest.mark.generic
@ pytest.mark.parametrize('config', ['history', 'nohistory'], indirect=True)
def test_console_advanced(config: str, dut: IdfDut) -> None:
    if config == 'history':
        dut.expect('Command history enabled')
    elif config == 'nohistory':
        dut.expect('Command history disabled')
```

To demonstrate how pytest is typically used in an ESP-IDF test script, let us go through this simple test script line by line in the following subsections.

**Target Markers**  Pytest markers can be used to indicate which targets (i.e., which ESP chip) a particular test case should run on. For example:

```python
@ pytest.mark.esp32    # <-- support esp32
@ pytest.mark.esp32c3  # <-- support esp32c3
@ pytest.mark.generic  # <-- test env "generic"
```

The example above indicates that a particular test case is supported on the ESP32 and ESP32-C3. Furthermore, the target’s board type should be generic. For more details regarding the generic type, you may run pytest --markers to get detailed information regarding all markers.

**Note:** If the test case can be run on all targets officially supported by ESP-IDF (call idf.py --list-targets for more details), you can use a special marker supported_targets to apply all of them in one line.

**Parameterized Markers**  You can use pytest.mark.parametrize with config to apply the same test to different apps with different sdkconfig files. For more information about sdkconfig.ci.xxx files, please refer to the Configuration Files section under this readme.

```python
@ pytest.mark.parametrize('config', ['history', 'nohistory'], indirect=True)  # <-- `indirect=True` is required
```

Overall, this test function would be replicated to 4 test cases:

- esp32.history.test_console_advanced
- esp32.nohistory.test_console_advanced
- esp32c3.history.test_console_advanced
- esp32c3.nohistory.test_console_advanced

**Testing Serial Output**  To ensure that test has executed successfully on target, the test script can test that serial output of the target using the dut.expect() function, for example:

```python
def test_console_advanced(config: str, dut: IdfDut) -> None:  # The value of _
    config = 'history':
        dut.expect('Command history enabled')
    elif config == 'nohistory':
        dut.expect('Command history disabled')
```

Espressif Systems  2190  Release master

Submit Document Feedback
Chapter 8. Contributions Guide

The `dut.expect(...)` will first compile the expected string into regex, which in turn is then used to seek through the serial output until the compiled regex is matched, or until a timeout occurs.

Please pay extra attention to the expected string when it contains regex keyword characters (e.g., parentheses, square brackets). Alternatively, you may use `dut.expect_exact(...)` that will attempt to match the string without converting it into regex.

For more information regarding the different types of `expect` functions, please refer to the `pytest-embedded Expecting documentation`.

Advanced Examples

Multi-Target Tests with the Same App  In some cases a test may involve multiple targets running the same test app. In this case, multiple DUTs can be instantiated using `parameterize`, for example:

```python
@ pytest.mark.esp32
@ pytest.mark.esp32s3
@ pytest.mark.usb_host
@ pytest.mark.parametrize('count', [2, ...])
def test_usb_host(dut: Tuple[IdfDut, IdfDut]) -> None:
    device = dut[0]  # <-- assume the first dut is the device
    host = dut[1]    # <-- and the second dut is the host
...
```

After setting the `count` to 2, all these fixtures are changed into tuples.

Multi-Target Tests with Different Apps  In some cases (in particular protocol tests), a test may involve multiple targets running different test apps (e.g., separate targets to act as master and slave). In this case, multiple DUTs with different test apps can be instantiated using `parameterize`.

This code example is taken from `pytest_wifi_getting_started.py`.

```python
@ pytest.mark.esp32
@ pytest.mark.multi_dut_generic
@ pytest.mark.parametrize(
    'count, app_path', [(2, f'{os.path.join(os.path.dirname(__file__), "softAP")}|{os.path.join(os.
ceptive_dirname(__file__), "station")}', ...])
)
def test_wifi_getting_started(dut: Tuple[IdfDut, IdfDut]) -> None:
    softap = dut[0]
    station = dut[1]
...
```

Here the first DUT was flashed with the app `softAP`, and the second DUT was flashed with the app `station`.

**Note:** Here the `app_path` should be set with absolute path. The `__file__` macro in Python would return the absolute path of the test script itself.

Multi-Target Tests with Different Apps and Targets This code example is taken from `pytest_wifi_getting_started.py`. As the comment says, for now it is not running in the ESP-IDF CI.
@pytest.mark.parametrize(
    'count, app_path, target', [
        (2,
         'os.path.join(os.path.dirname(__file__), "softAP") || os.path.join(os.
         path.dirname(__file__), "station")',
         'esp32\esp32s2'),
        (2,
         'os.path.join(os.path.dirname(__file__), "softAP") || os.path.join(os.
         path.dirname(__file__), "station")',
         'esp32s2\esp32'),
    ], indirect=True,
)
def test_wifi_getting_started(dut: Tuple[IdfDut, IdfDut]) -> None:
    softap = dut[0]
    station = dut[1]
    ...

Overall, this test function would be replicated to 2 test cases:

- softAP with ESP32 target, and station with ESP32-S2 target
- softAP with ESP32-S2 target, and station with ESP32 target

Support Different Targets with Different sdkconfig Files

This code example is taken from `pytest_panic.py` as an advanced example.

```python
CONFIGS = [
    pytest.param('coredump_flash_bin_crc', marks=[pytest.mark.esp32, pytest.mark.
    esp32s2]),
    pytest.param('coredump_flash_elf_sha', marks=[pytest.mark.esp32]), # sha256
    only supported on esp32
    pytest.param('coredump_uart_bin_crc', marks=[pytest.mark.esp32, pytest.mark.
    esp32s2]),
    pytest.param('coredump_uart_elf_crc', marks=[pytest.mark.esp32, pytest.mark.
    esp32s2]),
    pytest.param('gdbstub', marks=[pytest.mark.esp32, pytest.mark.esp32s2]),
    pytest.param('panic', marks=[pytest.mark.esp32, pytest.mark.esp32s2]),
]
@ pytest.mark.parametrize('config', CONFIGS, indirect=True)
```

Custom Classes

Usually, you may want to write a custom class under these conditions:

1. Add more reusable functions for a certain number of DUTs.
2. Add custom setup and teardown functions in different phases described in Section `pytest Execution Process`.

This code example is taken from `panic/conf/test.py`.

```python
class PanicTestDut(IdfDut):
    ...
@ pytest.fixture(scope='module')
def monkeypatch_module(request: FixtureRequest) -> MonkeyPatch:
    mp = MonkeyPatch()
    request.addfinalizer(mp.undo)
    return mp
@ pytest.fixture(scope='module', autouse=True)
```

(continues on next page)
def replace_dut_class(monkeypatch_module: MonkeyPatch) -> None:
    monkeypatch_modulesetattr('pytest_embedded_idf.dut.IdfDut', PanicTestDut)

monkeypatch_module provides a module-scoped monkeypatch fixture.
replace_dut_class is a module-scoped autouse fixture. This function replaces the IdfDut class with your custom class.

Mark Flaky Tests  Certain test cases are based on Ethernet or Wi-Fi. However, the test may be flaky due to networking issues. Thus, it is possible to mark a particular test case as flaky.
This code example is taken from pytest_esp_eth.py:

```python
@pytest.mark.flaky(reruns=3, reruns_delay=5)
def test_esp_eth_ip101(dut: IdfDut) -> None:
    ...
```

This flaky marker means that if the test function failed, the test case would rerun for a maximum of 3 times with 5 seconds delay.

Mark Known Failures  Sometimes, a test can consistently fail for the following reasons:
- The feature under test (or the test itself) has a bug.
- The test environment is unstable (e.g., due to network issues) leading to a high failure ratio.

Now you may mark this test case with marker xfail with a user-friendly readable reason.
This code example is taken from pytest_panic.py:

```python
@ pytest.mark.xfail('config.getvalue("target") == "esp32s2", reason='raised_-
→ IllegalInstruction instead')
def test_cache_error(dut: PanicTestDut, config: str, test_func_name: str) -> None:
    ...
```

This marker means that test is a known failure on the ESP32-S2.

Mark Nightly Run Test Cases  Some test cases are only triggered in nightly run pipelines due to a lack of runners.
This marker means that the test case would only be run with env var NIGHTLY_RUN or INCLUDE_NIGHTLY_RUN.

Mark Temporarily Disabled in CI  Some test cases which can pass locally may need to be temporarily disabled in CI due to a lack of runners.
This marker means that the test case could still be run locally with pytest --target esp32, but will not run in CI.

Run Unity Test Cases  For component-based unit test apps, all single-board test cases (including normal test cases and multi-stage test cases) can be run using the following command:

```python
def test_component_ut(dut: IdfDut):
    dut.run_all_single_board_cases()
```

Using this command will skip all the test cases containing the [ignore] tag.
If you need to run a group of test cases, you may run:
def test_component_ut(dut: IdfDut):
    dut.run_all_single_board_cases(group='psram')

It would trigger all test cases with the [psram] tag.

You may also see that there are some test scripts with the following statements, which are deprecated. Please use the suggested one as above.

def test_component_ut(dut: IdfDut):
    dut.expect_exact('Press ENTER to see the list of tests')
    dut.write('*')
    dut.expect_unity_test_output()

For further reading about our unit testing in ESP-IDF, please refer to our unit testing guide.

Running Tests in CI

The workflow in CI is simple, build jobs > target test jobs.

Build Jobs

Build Job Names

- Component-based Unit Tests: build_pytest_components_<target>
- Example Tests: build_pytest_examples_<target>
- Custom Tests: build_pytest_test_apps_<target>

Build Job Commands The command used by CI to build all the relevant tests is: python $IDF_PATH/tools/ci/ci_build_apps.py <parent_dir> --target <target> -vv --pytest-apps

All apps which supported the specified target would be built with all supported sdkconfig files under build_<target>_<config>.

For example, if you run python $IDF_PATH/tools/ci/ci_build_apps.py $IDF_PATH/examples/system/console/basic --target esp32 --pytest-apps, the folder structure would be like this:

```
basic
├── build_esp32_history/
│   ├── ...
│ └── build_esp32_nohistory/
│   ├── ...
└── main/
    ├── CMakeLists.txt
    └── pytest_console_basic.py
```

All the build folders would be uploaded as artifacts under the same directories.

Target Test Jobs

Target Test Job Names

- Component-based Unit Tests: component_ut_pytest_<target>_<test_env>
- Example Tests: example_test_pytest_<target>_<test_env>
- Custom Tests: test_app_test_pytest_<target>_<test_env>
Chapter 8. Contributions Guide

Target Test Job Commands  The command used by CI to run all the relevant tests is: `pytest <parent_dir> --target <target> -m <test_env_marker>`

All test cases with the specified target marker and the test env marker under the parent folder would be executed.

The binaries in the target test jobs are downloaded from build jobs. The artifacts would be placed under the same directories.

Running Tests Locally

First you need to install ESP-IDF with additional Python requirements:

```bash
$ cd $IDF_PATH
$ bash install.sh --enable-pytest
$ ./export.sh
```

By default, the pytest script will look for the build directory in this order:

- `build_<target>_<sdkconfig>`
- `build_<target>`
- `build_<sdkconfig>`
- `build`

Which means, the simplest way to run pytest is calling `idf.py build`.

For example, if you want to run all the esp32 tests under the `$IDF_PATH/examples/get-started/hello_world` folder, you should run:

```bash
$ cd examples/get-started/hello_world
$ idf.py build
$ pytest --target esp32
```

If you have multiple sdkconfig files in your test app, like those `sdkconfig.ci.*` files, the simple `idf.py build` won’t apply the extra sdkconfig files. Let us take `$IDF_PATH/examples/system/console/basic` as an example.

If you want to test this app with config history, and build with `idf.py build`, you should run:

```bash
$ cd examples/system/console/basic
$ idf.py -DSDKCONFIG_DEFAULTS="sdkconfig.defaults;sdkconfig.ci.history" build
$ pytest --target esp32 --sdkconfig history
```

If you want to build and test with all sdkconfig files at the same time, you should use our CI script as an helper script:

```bash
$ cd examples/system/console/basic
$ python $IDF_PATH/tools/ci/ci_build_apps.py --target esp32 -vv --pytest-apps
$ pytest --target esp32
```

The app with `sdkconfig.ci.history` will be built in `build_esp32_history`, and the app with `sdkconfig.ci.nohistory` will be built in `build_esp32_nohistory`. `pytest --target esp32` will run tests on both apps.

Tips and Tricks

Filter the Test Cases

- Filter by target with `pytest --target <target>`
  pytest would run all the test cases that support specified target.
- Filter by sdkconfig file with `pytest --sdkconfig <sdkconfig>`
  If `<sdkconfig>` is default, pytest would run all the test cases with the sdkconfig file `sdkconfig.defaults`.
  In other cases, pytest would run all the test cases with sdkconfig file `sdkconfig.ci.<sdkconfig>`. 
Add New Markers  We are using two types of custom markers, target markers which indicate that the test cases should support this target, and env markers which indicate that the test cases should be assigned to runners with these tags in CI.

You can add new markers by adding one line under the `${IDF_PATH}/conftest.py`. If it is a target marker, it should be added into TARGET_MARKERS. If it is a marker that specifies a type of test environment, it should be added into ENV_MARKERS. The syntax should be: `<marker_name>: <marker_description>`.

Generate JUnit Report  You can call pytest with `--junitxml <filepath>` to generate the JUnit report. In ESP-IDF, the test case name would be unified as `<target>.<config>.<function_name`.

Skip Auto Flash Binary  Skipping auto-flash binary every time would be useful when you are debugging your test script.

You can call pytest with `--skip-autoflash y` to achieve it.

Record Statistics  Sometimes you may need to record some statistics while running the tests, like the performance test statistics.

You can use `record_xml_attribute` fixture in your test script, and the statistics would be recorded as attributes in the JUnit report.

Logging System  Sometimes you may need to add some extra logging lines while running the test cases.

You can use Python logging module to achieve this.

Useful Logging Functions (as Fixture)

**log_performance**

```python
def test_hello_world(
    dut: IdfDut,
    log_performance: Callable[[str, object], None],
) -> None:
    log_performance('test', 1)
```

The above example would log the performance item with pre-defined format: `[performance][test]: 1` and record it under the `properties` tag in the JUnit report if `--junitxml <filepath>` is specified. The JUnit test case node would look like:

```xml
<testcase classname="examples.get-started.hello_world.pytest_hello_world" file="~examples/get-started/hello_world/pytest_hello_world.py" line="13" name="esp32.default.test_hello_world" time="8.389">  
  <properties>
    <property name="test" value="1"/>
  </properties>
</testcase>
```

**check_performance**  We provide C macros `TEST_PERFORMANCE_LESS_THAN` and `TEST_PERFORMANCE_GREATER_THAN` to log the performance item and check if the value is in the valid range. Sometimes the performance item value could not be measured in C code, so we also provide a Python function for the same purpose. Please note that using C macros is the preferred approach, since the Python function could not recognize the threshold values of the same performance item under different ifdef blocks well.
def test_hello_world(
    dut: IdfDut,
    check_performance: Callable[[str, float, str], None],
) -> None:
    check_performance('RSA_2048KEY_PUBLIC_OP', 123, 'esp32')
    check_performance('RSA_2048KEY_PUBLIC_OP', 19001, 'esp32')

The above example would first get the threshold values of the performance item RSA_2048KEY_PUBLIC_OP from components/idf_test/include/idf_performance.h and the target-specific one components/idf_test/include/esp32/idf_performance_target.h, then check if the value reached the minimum limit or exceeded the maximum limit.

Let us assume the value of IDF_PERFORMANCE_MAX_RSA_2048KEY_PUBLIC_OP is 19000. so the first check_performance line would pass and the second one would fail with warning: [Performance] RSA_2048KEY_PUBLIC_OP value is 19001, doesn\'t meet pass standard 19000.0.

Further Readings

- pytest documentation: https://docs.pytest.org/en/latest/contents.html
- pytest-embedded documentation: https://docs.espressif.com/projects/pytest-embedded/en/latest/
Chapter 9

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.

9.1 Releases

The documentation for the current stable release version can always be found at this URL:


Documentation for the latest version (master branch) can always be found at this URL:


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.

9.2 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. Also refer to Compatibility Between ESP-IDF Releases and Revisions of Espressif SoCs to make sure the ESP-IDF version you selected is compatible with the chip revision you are going to produce with.
- 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.
- If you plan to use another project which is based on ESP-IDF, please check the documentation of that project to determine the version(s) of ESP-IDF it is compatible with.

See Updating ESP-IDF if you already have a local copy of ESP-IDF and wish to update it.
9.3 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.

9.4 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 bugfix 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.

Each ESP-IDF major and minor release (V4.1, V4.2, etc) is supported for 30 months after the initial stable release date.

Supported means that the ESP-IDF team will continue to apply bug fixes, security fixes, etc to the release branch on GitHub, and periodically make new bugfix releases as needed.

Support period is divided into “Service” and “Maintenance” period:

<table>
<thead>
<tr>
<th>Period</th>
<th>Duration</th>
<th>Recommended for new projects?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>12 months</td>
<td>Yes</td>
</tr>
<tr>
<td>Maintenance</td>
<td>18 months</td>
<td>No</td>
</tr>
</tbody>
</table>

During the Service period, bugfixes releases are more frequent. In some cases, support for new features may be added during the Service period (this is reserved for features which are needed to meet particular regulatory requirements or standards for new products, and which carry a very low risk of introducing regressions.)

During the Maintenance period, the version is still supported but only bugfixes for high severity issues or security issues will be applied.

Using an “In Service” version is recommended when starting a new project.
Users are encouraged to upgrade all projects to a newer ESP-IDF release before the support period finishes and the release becomes End of Life (EOL). It is our policy to not continue fixing bugs in End of Life releases.

Pre-release versions (betas, previews, -rc and -dev versions, etc) are not covered by any support period. Sometimes a particular feature is marked as “Preview” in a release, which means it is also not covered by the support period.

The ESP-IDF Programming Guide has information about the different versions of ESP-IDF (major, minor, bugfix, etc).

### 9.5 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:
## Chapter 9. ESP-IDF Versions

<table>
<thead>
<tr>
<th>Version String</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 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 release branch).  
| | - 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. |

### 9.6 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.

### 9.7 Updating ESP-IDF

Updating ESP-IDF depends on which version(s) you wish to follow:

- *Updating to Stable Release* is recommended for production use.
Chapter 9. ESP-IDF Versions

- 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.

### 9.7.1 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.
- In Linux or macOS system, please run the following commands to update the local branch to vX.Y.Z:

```bash
cd $IDF_PATH
git fetch
git checkout vX.Y.Z
git submodule update --init --recursive
```

- In the Windows system, please replace cd $IDF_PATH with cd %IDF_PATH%.
- 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.

### 9.7.2 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.

### 9.7.3 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:

- In Linux or macOS system, please run the following commands to check out to the master branch locally:

```bash
cd $IDF_PATH
git checkout master
git pull
git submodule update --init --recursive
```

- In the Windows system, please replace cd $IDF_PATH with cd %IDF_PATH%.
- 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 unsupportable “snapshots” that may have undocumented bugs. For a semi-stable version, try *Updating to a Release Branch* instead.

### 9.7.4 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, in Linux or macOS system, you can execute the following commands to follow the branch for ESP-IDF v3.1, including any bugfixes for future releases like v3.1.1, etc:

```bash
cd $IDF_PATH
git fetch
git checkout release/v3.1
git pull
git submodule update --init --recursive
```

In the Windows system, please replace `cd $IDF_PATH` with `cd %IDF_PATH%`.

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 10

Resources

10.1 PlatformIO

10.1.1 What Is PlatformIO?

PlatformIO is a cross-platform embedded development environment with out-of-the-box support for ESP-IDF.

Since ESP-IDF support within PlatformIO is not maintained by the Espressif team, please report any issues with PlatformIO directly to its developers in the official PlatformIO repositories.

A detailed overview of the PlatformIO ecosystem and its philosophy can be found in the official PlatformIO documentation.

10.1.2 Installation

- **PlatformIO IDE** is a toolset for embedded C/C++ development available on Windows, macOS and Linux platforms.
- **PlatformIO Core (CLI)** is a command-line tool that consists of multi-platform build system, platform and library managers and other integration components. It can be used with a variety of code development environments and allows integration with cloud platforms and web services.
10.1.3 Configuration

Please go through the official PlatformIO configuration guide for ESP-IDF.

10.1.4 Tutorials

- ESP-IDF and ESP32-DevKitC: debugging, unit testing, project analysis

10.1.5 Project Examples

Please check ESP-IDF page in the official PlatformIO documentation

10.1.6 Next Steps

Here are some useful links for exploring the PlatformIO ecosystem:

- Learn more about integrations with other IDEs or Text Editors
- Get help from PlatformIO community

10.2 CLion

10.2.1 What Is CLion?

CLion is a cross-platform integrated Development Environment (IDE) for C and C++ programming. CLion also provides dedicated support for ESP-IDF, allowing developers to seamlessly work with the ESP-IDF framework.

10.2.2 Installation

To install CLion, please follow the instructions provided in Install CLion for your operating system (Windows, macOS, or Linux).

10.2.3 Configuration

To configure an ESP-IDF project in CLion, please refer to the guide on Configure an ESP CMake project in CLion. This guide will walk you through the necessary steps to set up your project properly.

10.2.4 Resources

For more information about CLion and ESP-IDF integration, please refer to the following resource:

- CLion Documentation: The official documentation for CLion provides detailed information on various aspects of the IDE, including ESP-IDF integration.

10.3 VisualGDB

10.3.1 What Is VisualGDB?

VisualGDB is a powerful extension for Microsoft Visual Studio that provides advanced development tools and features for embedded systems, including support for the ESP-IDF framework. VisualGDB allows you to leverage the familiar
and feature-rich Visual Studio environment for your ESP-IDF projects, enabling efficient coding, debugging, and deployment.

10.3.2 Installation

Please download and install VisualGDB by following the steps stated in VisualGDB download and installation.

10.3.3 Configuration

Creating Advanced ESP32 Projects with ESP-IDF provide basic steps about how to configure an ESP-IDF project in VisualGDB.

You can also refer to Advanced ESP-IDF Project Structure to get a more comprehensive impression for developing ESP-IDF projects using VisualGDB.

10.3.4 Resources

For more information about VisualGDB and ESP-IDF integration, refer to the following resources:

- VisualGDB Documentation: The official documentation for VisualGDB provides comprehensive guides and tutorials on using VisualGDB with ESP-IDF.

For inquiries related to these third-party tools, we recommend seeking assistance from the respective tool’s support channels or user communities.

10.4 Useful Links

- 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 ESP 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 the ESP32, ESP32-S2 and ESP32-C3.
- 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-S2 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 11

Copyrights and Licenses

11.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2023 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.

Some examples use external components which are not Apache licensed, please check the copyright description in each example source code.

11.1.1 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) 2017 Amazon.com, Inc. or its affiliates are licensed under the MIT License, 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) 2003-2022 Jouni Malinen <j@w1.fi> and contributors and licensed under the BSD license.
- **Fast PBKDF2** Copyright (c) 2015 Joseph Birr-Pixton and licensed under CC0 Public Domain Dedication license.
- **FreeBSD net80211** Copyright (c) 2004-2008 Sam Leffler, Ernmo Consulting and licensed under the BSD license.
- **argtable3** argument parsing library Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann and licensed under 3-clause BSD license. argtable3 also includes the following software components. For details, please see argtable3 LICENSE file.
  - C Hash Table library, Copyright (c) 2002, Christopher Clark and licensed under 3-clause BSD license.
  - The Better String library, Copyright (c) 2014, Paul Hsieh and licensed under 3-clause BSD license.
  - TCL library, Copyright the Regents of the University of California, Sun Microsystems, Inc., Scrippts Corporation, ActiveState Corporation and other parties, and licensed under TCL/TK License.
- **linenoise** line editing library Copyright (c) 2010-2014 Salvatore Sanfilippo, Copyright (c) 2010-2013 Pieter Noordhuis, licensed under 2-clause BSD 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 as described in LICENSE file.
• micro-ec library, Copyright (c) 2014 Kenneth MacKay, is licensed under 2-clause BSD license.
• Mbed TLS library, Copyright (C) 2006-2018 ARM Limited, is licensed under Apache License 2.0 as described in LICENSE file.
• SPIFFS library, Copyright (c) 2013-2017 Peter Andersson, is licensed under MIT license as described in LICENSE file.
• SD/MMC driver is derived from OpenBSD SD/MMC driver, Copyright (c) 2006 Uwe Stuehler, and is licensed under BSD license.
• ESP-MQTT MQTT Package (contiki-mqtt) - Copyright (c) 2014, Stephen Robinson, MQTT-ESP - Tuan PM <tuannpm at live dot com> is licensed under Apache License 2.0 as described in LICENSE file.
• 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 as described in LICENSE file.
• TLSF allocator Two Level Segregated Fit memory allocator, Copyright (c) 2006-2016, Matthew Conte, and licensed under the BSD 3-clause license.
• openthread, Copyright (c) The OpenThread Authors, is licensed under BSD License as described in LICENSE file.
• UBSAN runtime — Copyright (c) 2016, Linaro Limited and Jiří Zárevůčky, licensed under the BSD 2-clause license.
• HTTP Parser Based on src/http/ngx_http_parse.c from NGINX copyright Igor Sysoev. Additional changes are licensed under the same terms as NGINX and Joyent, Inc. and other Node contributors. For details please check LICENSE file.
• SEGGER SystemView target-side library, Copyright (c) 1995-2021 SEGGER Microcontroller GmbH, is licensed under BSD 1-clause license.

11.1.2 Documentation

• HTML version of the ESP-IDF Programming Guide uses the Sphinx theme sphinx_idf_theme, which is Copyright (c) 2013-2020 Dave Snider, Read the Docs, Inc. & contributors, and Espressif Systems (Shanghai) CO., LTD. It is based on sphinx_rtd_theme. Both are licensed under MIT license.

11.2 ROM Source Code Copyrights

Espressif SoCs 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.
• TJpgDec Copyright (C) 2011, ChaN, all right reserved. See below for license.
• Parts of Zephyr RTOS USB stack:
  – DesignWare USB device driver Copyright (c) 2016 Intel Corporation and licensed under Apache 2.0 license.
  – Generic USB device driver Copyright (c) 2006 Bertrik Sikken (bertrik@sikken.nl), 2016 Intel Corporation and licensed under BSD 3-clause license.
  – USB descriptors functionality Copyright (c) 2017 PHYTEC Messtechnik GmbH, 2017-2018 Intel Corporation and licensed under Apache 2.0 license.
  – USB DFU class driver Copyright (c) 2015-2016 Intel Corporation, 2017 PHYTEC Messtechnik GmbH and licensed under BSD 3-clause license.
  – USB CDC ACM class driver Copyright (c) 2015-2016 Intel Corporation and licensed under Apache 2.0 license.
11.3  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.

11.4  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.

11.5  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 12

About

This is documentation of ESP-IDF, the framework to develop applications for ESP32-S2. The ESP32-S2 is a 2.4 GHz Wi-Fi SoC, which integrates a Xtensa® 32-bit LX7 CPU.

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-S2 using Windows, Linux and macOS operating systems.
Chapter 13

Switch Between Languages

The ESP-IDF Programming Guide is now available in two languages. Please refer to the English version if there is any discrepancy.

• English
• Chinese

You can easily change from one language to another by clicking the language link you can find at the top of every document that has a translation.
Symbols

_index (C macro), 1642
_ip_addr (C++ struct), 430
_ip_addr::ip4 (C++ member), 431
_ip_addr::type (C++ member), 431
_ip_addr::u_addr (C++ member), 431
[anonymous] (C++ enum), 707, 1705
[anonymous]::ESP_ERR_FLASH_NO_RESPONSE (C++ enumerator), 707
[anonymous]::ESP_ERR_FLASH_SIZE_NOT_MATCH (C++ enumerator), 707
[anonymous]::ESP_ERR_SLEEP_REJECT (C++ enumerator), 1705
[anonymous]::ESP_ERR_SLEEP_TOO_SHORT_SLEEP_DURATION (C++ enumerator), 1705

A

adc_atten_t (C++ enum), 445
adc_atten_t::ADC_ATTEN_DB_0 (C++ enumerator), 445
adc_atten_t::ADC_ATTEN_DB_11 (C++ enumerator), 445
adc_atten_t::ADC_ATTEN_DB_12 (C++ enumerator), 445
adc_atten_t::ADC_ATTEN_DB_2_5 (C++ enumerator), 445
adc_atten_t::ADC_ATTEN_DB_6 (C++ enumerator), 445
adc_bitwidth_t (C++ enum), 445
adc_bitwidth_t::ADC_BITWIDTH_10 (C++ enumerator), 445
adc_bitwidth_t::ADC_BITWIDTH_11 (C++ enumerator), 445
adc_bitwidth_t::ADC_BITWIDTH_12 (C++ enumerator), 446
adc_bitwidth_t::ADC_BITWIDTH_13 (C++ enumerator), 446
adc_bitwidth_t::ADC_BITWIDTH_9 (C++ enumerator), 445
adc_bitwidth_t::ADC_BITWIDTH_DEFAULT (C++ enumerator), 445
adc_cali_check_scheme (C++ function), 462
adc_cali_handle_t (C++ type), 462
adc_cali_raw_to_voltage (C++ function), 462
adc_cali_scheme_ver_t (C++ enum), 462
adc_cali_scheme_ver_t::ADC_CALI_SCHEME_VER_CURVE_FITTING (C++ enumerator), 462
adc_cali_scheme_ver_t::ADC_CALI_SCHEME_VER_LINE_FITTING (C++ enumerator), 462
adc_channel_t (C++ enum), 444
adc_channel_t::ADC_CHANNEL_0 (C++ enumerator), 444
adc_channel_t::ADC_CHANNEL_1 (C++ enumerator), 444
adc_channel_t::ADC_CHANNEL_2 (C++ enumerator), 444
adc_channel_t::ADC_CHANNEL_3 (C++ enumerator), 444
adc_channel_t::ADC_CHANNEL_4 (C++ enumerator), 444
adc_channel_t::ADC_CHANNEL_5 (C++ enumerator), 445
adc_channel_t::ADC_CHANNEL_6 (C++ enumerator), 445
adc_channel_t::ADC_CHANNEL_7 (C++ enumerator), 445
adc_channel_t::ADC_CHANNEL_8 (C++ enumerator), 445
adc_channel_t::ADC_CHANNEL_9 (C++ enumerator), 445
adc_continuous_callback_t (C++ type), 459
adc_continuous_channel_to_io (C++ function), 458
adc_continuous_clk_src_t (C++ type), 444
adc_continuous_config (C++ function), 456
adc_continuous_config (C++ struct), 458
adc_continuous_config_t::adc_pattern (C++ member), 458
adc_continuous_config_t::conv_mode (C++ member), 458
adc_continuous_config_t::format (C++ member), 458
adc_continuous_config_t::pattern_num (C++ member), 458
adc_continuous_config_t::sample_freq_hz (C++ member), 458
adc_continuous_deinit (C++ function), 457
adc_continuous_evt_cbs_t (C++ struct), 459
adc_continuous_evt_cbs_t::on_conv_done (C++ member), 459
adc_continuous_evt_cbs_t::on_pool_ovf (C++ member), 459
450
adc.oneshot_unit_init_cfg_t::clk_src
(C++ member), 450
adc.oneshot_unit_init_cfg_t::ulp_mode
(C++ member), 450
adc.oneshot_unit_init_cfg_t::unit_id
(C++ member), 450
adc.ulp_mode_t (C++ enum), 446
adc.ulp_mode_t::ADC_ULP_MODE_DISABLE
(C++ enumerator), 446
adc.ulp_mode_t::ADC_ULP_MODE_FSM
(C++ enumerator), 446
adc.ulp_mode_t::ADC_ULP_MODE_RISCV
(C++ enumerator), 446
adc_unit_t (C++ enum), 444
adc_unit_t::ADC_UNIT_1
(C++ enumerator), 444
adc_unit_t::ADC_UNIT_2
(C++ enumerator), 444
ALU_SEL_ADD (C macro), 1756
ALU_SEL_AND (C macro), 1756
ALU_SEL_LSH (C macro), 1757
ALU_SEL_MOV (C macro), 1757
ALU_SEL_OR (C macro), 1756
ALU_SEL_RSH (C macro), 1757
ALU_SEL_STAGE_DEC (C macro), 1757
ALU_SEL_STAGE_INC (C macro), 1757
ALU_SEL_STAGE_RST (C macro), 1757
ALU_SEL_STAGE_RST (C macro), 1757
async_memcpy_config_t (C++ struct), 1729
async_memcpy_config_t::backlog
(C++ member), 1729
async_memcpy_config_t::flags
(C++ member), 1729
async_memcpy_config_t::psram_trans_align
(C++ member), 1729
async_memcpy_config_t::sramp_trans_align
(C++ member), 1729
ASYNC_MEMCPY_DEFAULT_CONFIG
(C macro), 1729
async_memcpy_event_t (C++ struct), 1729
async_memcpy_event_t::data
(C++ member), 1729
async_memcpy_handle_t (C++ type), 1729
async_memcpy_isr_cb_t (C++ type), 1729

B
B_CMP_E (C macro), 1757
B_CMP_G (C macro), 1757
B_CMP_LE (C macro), 1758
BLE_UUID128_VAL_LENGTH (C macro), 1239
bootsloader_fill_random (C++ function), 1690
bootsloader_random_disable (C++ function), 1690
bootsloader_random_enable (C++ function), 1690
bridgeif_config (C++ struct), 423
bridgeif_config::max_fdb_dyn_entries
(C++ member), 423
bridgeif_config::max_fdb_sta_entries
(C++ member), 423
bridgeif_config::max_ports
(C++ member), 423
bridgeif_config_t (C++ type), 426
BS_CMP_GE (C macro), 1758
BS_CMP_L (C macro), 1758
bs_cmp_le (C macro), 1758
btm_query_reason (C++ enum), 346
btm_query_reason::REASON_BANDWIDTH
(C++ enumerator), 347
btm_query_reason::REASON_DELAY
(C++ enumerator), 347
btm_query_reason::REASON_FRAME_LOSS
(C++ enumerator), 347
btm_query_reason::REASON_GRAY_ZONE
(C++ enumerator), 347
btm_query_reason::REASON_INTERFERENCE
(C++ enumerator), 347
btm_query_reason::REASON_LOAD_BALANCE
(C++ enumerator), 347
btm_query_reason::REASON_PREMIUM_AP
(C++ enumerator), 347
btm_query_reason::REASON_RETRANSCTIONS
(C++ enumerator), 347
btm_query_reason::REASON_RSSI
(C++ enumerator), 347
btm_query_reason::REASON_UNSPECIFIED
(C++ enumerator), 347
BX_JUMP_TYPE_DIRECT (C macro), 1757
BX_JUMP_TYPE_OVF (C macro), 1757
BX_JUMP_TYPE_ZERO (C macro), 1757

C
CHIP_FEATURE_BLE (C macro), 1654
CHIP_FEATURE_BT (C macro), 1654
CHIP_FEATURE_EMB_FLASH (C macro), 1654
CHIP_FEATURE_EMB_PSRAM (C macro), 1654
CHIP_FEATURE_IEEE802154 (C macro), 1654
CHIP_FEATURE_WIFI_BGN (C macro), 1654
CONFIG_FEATURE_CACHE_TX_BUF_BIT
(C macro), 293
CONFIG_FEATURE_FTM_TX_BUF_BIT
(C macro), 293
CONFIG_FEATURE_FTM_INITIATOR_BIT
(C macro), 293
CONFIG_FEATURE_FTM_RESPONDER_BIT
(C macro), 293
CONFIG_FEATURE_WPA3_SAE_BIT
(C macro), 293
CONFIG_HEAP_TRACING_STACK_DEPTH
(C macro), 1620

D
dac_channel_mask_t (C++ enum), 486
dac_channel_mask_t::DAC_CHANNEL_MASK_ALL
(C++ enumerator), 486
dac_channel_mask_t::DAC_CHANNEL_MASK_CH0 (C++ enumerator), 486
dac_channel_mask_t::DAC_CHANNEL_MASK_CH1 (C++ enumerator), 486
dac_channel_t (C++ enum), 487
dac_channel_t::DAC_CHAN_0 (C++ enumerator), 487
dac_channel_t::DAC_CHAN_1 (C++ enumerator), 487
dac_channel_t::DAC_CHANNEL_0 (C++ enum), 487
dac_channel_t::DAC_CHANNEL_1 (C++ enum), 487
dac_continuous_channel_mode_t (C++ enum), 486
dac_continuous_channel_mode_t::DAC_CHANNEL_MODE_CONFIG (C++ struct), 487

dac_continuous_config_t (C++ struct), 484
dac_continuous_config_t::buf_size (C++ member), 484
dac_continuous_config_t::chan_mask (C++ member), 484
dac_continuous_config_t::chan_mode (C++ member), 485
dac_continuous_config_t::clk_src (C++ member), 485
dac_continuous_config_t::desc_num (C++ member), 484
dac_continuous_config_t::freq_hz (C++ member), 484
dac_continuous_config_t::offset (C++ member), 484
dac_continuous_config_t::on_stop (C++ function), 481
dac_continuous_config_t::on_convert_done (C++ function), 481
dac_continuous_digi_clk_src_t (C++ type), 486
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_free (C++ function), 482
dac_continuous_new_channels (C++ function), 485
dac_continuous_register_event_callback (C++ function), 483
dac_continuous_start_async_write (C++ function), 483
dac_continuous_stop_async_write (C++ function), 483
dac_continuous_write (C++ function), 481
dac_continuous_write_asynchronously (C++ function), 483
dac_continuous_write_cyclically (C++ function), 482
dac_continuous_atten_t (C++ enum), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DEFAULT (C++ enumerator), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DEFAULT (C++ enumerator), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_18 (C++ enumerator), 488
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_6 (C++ enumerator), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DEFAULT (C++ enumerator), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DEFAULT (C++ enumerator), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_12 (C++ enumerator), 488
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_0 (C++ enumerator), 486
dac_continuous_config_t::atts (C++ member), 480

dac_continuous_config_t::chan_id (C++ member), 480
dac_continuous_config_t::clk_src (C++ member), 480
dac_continuous_config_t::flags (C++ member), 480

dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_new_channels (C++ function), 485
dac_continuous_register_event_callback (C++ function), 483
dac_continuous_start_async_write (C++ function), 483
dac_continuous_stop_async_write (C++ function), 483
dac_continuous_write (C++ function), 481
dac_continuous_write_asynchronously (C++ function), 483
dac_continuous_write_cyclically (C++ function), 482
dac_continuous_atten_t (C++ enum), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_12 (C++ enum), 487

dac_continuous_config_t::atts (C++ member), 480
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_new_channels (C++ function), 485
dac_continuous_register_event_callback (C++ function), 483
dac_continuous_start_async_write (C++ function), 483
dac_continuous_stop_async_write (C++ function), 483
dac_continuous_write (C++ function), 481
dac_continuous_write_asynchronously (C++ function), 483
dac_continuous_write_cyclically (C++ function), 482
dac_continuous_atten_t (C++ enum), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_12 (C++ enum), 487

dac_continuous_config_t::atts (C++ member), 480
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_new_channels (C++ function), 485
dac_continuous_register_event_callback (C++ function), 483
dac_continuous_start_async_write (C++ function), 483
dac_continuous_stop_async_write (C++ function), 483
dac_continuous_write (C++ function), 481
dac_continuous_write_asynchronously (C++ function), 483
dac_continuous_write_cyclically (C++ function), 482
dac_continuous_atten_t (C++ enum), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_12 (C++ enum), 487

dac_continuous_config_t::atts (C++ member), 480
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_enable (C++ function), 481
dac_continuous_new_channels (C++ function), 485
dac_continuous_register_event_callback (C++ function), 483
dac_continuous_start_async_write (C++ function), 483
dac_continuous_stop_async_write (C++ function), 483
dac_continuous_write (C++ function), 481
dac_continuous_write_asynchronously (C++ function), 483
dac_continuous_write_cyclically (C++ function), 482
dac_continuous_atten_t (C++ enum), 487
dac_continuous_atten_t::DAC_COSINE_ATTEN_DB_12 (C++ enum), 487

Espressif Systems 2220 Release master

Submit Document Feedback
Index

(C++ member), 530
dedic_gpio_bundle_config_t::flags
(C++ member), 531
dedic_gpio_bundle_config_t::gpio_array
(C++ member), 530
dedic_gpio_bundle_config_t::in_en
(C++ member), 530
dedic_gpio_bundle_config_t::in_invert
(C++ member), 530
dedic_gpio_bundle_config_t::out_en
(C++ member), 530
dedic_gpio_bundle_config_t::out_invert
(C++ member), 530
dedic_gpio_bundle_config_t::out_type
(C++ member), 531
dedic_gpio_bundle_read_in (C++ function),
529
dedic_gpio_bundle_read_out (C++ function),
529
dedic_gpio_bundle_set_interrupt_and_callback
(C++ function), 529
dedic_gpio_bundle_write (C++ function), 529
dedic_gpio_del_bundle (C++ function), 529
dedic_gpio_get_in_mask (C++ function), 528
dedic_gpio_get_in_offset (C++ function), 528
dedic_gpio_get_out_mask (C++ function), 528
dedic_gpio_get_out_offset (C++ function), 528
dedic_gpio_intr_type_t (C++ enum), 531
dedic_gpio_intr_type_t::DEDIC_GPIO_INTR_BOTH_EDGE
(C++ enumerator), 531
dedic_gpio_intr_type_t::DEDIC_GPIO_INTR_HIGH_LEVEL
(C++ enumerator), 531
dedic_gpio_intr_type_t::DEDIC_GPIO_INTR_LOW_LEVEL
(C++ enumerator), 531
dedic_gpio_intr_type_t::DEDIC_GPIO_INTR_NEG_EDGE
(C++ enumerator), 531
dedic_gpio_intr_type_t::DEDIC_GPIO_INTR_POS_EDGE
(C++ enumerator), 531
dedic_gpio_intr_type_t::DEDIC_GPIO_INTR_BOTH_EDGE
(C++ enumerator), 531
dedic_gpio_isr_callback_t (C++ type), 531
dedic_gpio_new_bundle (C++ function), 528
DEFAULT_HTTP_BUF_SIZE (C macro), 142
dpp_bootstrap_type (C++ enum), 351
dpp_bootstrap_type::DPP_BOOTSTRAP_NFC_URI
(C++ enumerator), 351
dpp_bootstrap_type::DPP_BOOTSTRAP_PKEX
(C++ enumerator), 351
dpp_bootstrap_type::DPP_BOOTSTRAP_QR_CODE
(C++ enumerator), 351
efuse_hal_flash_encryption_enabled
(C++ function), 1374
efuse_hal_get_disable_wafer_version_major
(C++ function), 1374
efuse_hal_get_mac (C++ function), 1373
efuse_hal_get_major_chip_version (C++ function), 1374
efuse_hal_get_minor_chip_version (C++ function), 1374
emac_rmii_clock_gpio_t (C++ enum), 379
emac_rmii_clock_gpio_t::EMAC_APPL_CLK_OUT_GPI
(C++ enumerator), 379
emac_rmii_clock_gpio_t::EMAC_CLK_IN_GPIO
(C++ enumerator), 379
emac_rmii_clock_gpio_t::EMAC_CLK_OUT_180_GPIO
(C++ enumerator), 380
emac_rmii_clock_gpio_t::EMAC_CLK_OUT_GPIO
(C++ enumerator), 379
eVisit::rmii_clock_mode_t (C++ enum), 379
eVisit::rmii_clock_mode_t::EMAC_CLK_DEFAULT
(C++ enumerator), 379
eVisit::rmii_clock_mode_t::EMAC_CLK_EXT_IN
(C++ enumerator), 379
eVisit::rmii_clock_mode_t::EMACL_CLOCK_OUT
(C++ enumerator), 379
eNotifyAction (C++ enum), 1475
eNotifyAction::eSetValueWithOverwrite
(C++ function), 1476
eNotifyAction::eSetValueWithoutOverwrite
(C++ function), 1476
eNotifyAction::eNoAction
(C++ function), 1476
eNotifyAction::eSetBits
(C++ function), 1476
eNotifyAction::eIncrement
(C++ function), 1476
emac::sleep_mode_status::eAbortSleep
(C++ enum), 1476
eVisit::sleep_mode_status::eStandardSleep
(C++ enum), 1476
eVisit::sleep_mode_status::eStandardSleep
(C++ function), 1476
esp_alloc_failed_hook_t (C++ type), 1591
ESP_APP_DESC_MAGIC_WORD (C macro), 1661
esp_app_desc_t (C++ struct), 1591
esp_app_desc_t::app_elf_sha256
(C++ member), 1661
esp_app_desc_t::date (C++ member), 1661
esp_app_desc_t::idf_ver
(C++ member), 1661
esp_app_desc_t::magic_word
(C++ member), 1660
esp_app_desc_t::project_name
(C++ member), 1660
esp_app_desc_t::reserv1
(C++ member), 1660
esp_app_desc_t::reserv2
(C++ member), 1661
esp_app_desc_t::secure_version
(C++ member), 1660
efuse_hal_blk_version (C++ function), 1374
efuse_hal_chip_revision (C++ function), 1373
EFD_SUPPORT_ISR (C macro), 1352
Index

esp_app_desc_t::time (C++ member), 1660
esp_app_desc_t::version (C++ member), 1660
esp_app_get_description (C++ function), 1660
esp_app_get_elf_sha256 (C++ function), 1660
esp_app_get_elf_sha256_str (C++ function), 1660
esp_aptrace_buffer_get (C++ function), 1364
esp_aptrace_buffer_put (C++ function), 1365
esp_aptrace_dest_t (C++ enum), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_JTAG (C++ enumerator), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_MAX (C++ enumerator), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_NAND (C++ enumerator), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_NIC (C++ enumerator), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_UART (C++ enumerator), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_TRAX (C++ enumerator), 1368
esp_aptrace_dest_t::ESP_APTRACE_DEST_NUM (C++ enumerator), 1368
esp_aptrace_down_buffer_config (C++ function), 1364
esp_aptrace_down_buffer_get (C++ function), 1366
esp_aptrace_down_buffer_put (C++ function), 1366
esp_aptrace_fclose (C++ function), 1366
esp_aptrace_feof (C++ function), 1367
esp_aptrace_flush (C++ function), 1365
esp_aptrace_flush_nolock (C++ function), 1365
esp_aptrace_fopen (C++ function), 1366
esp_aptrace_fread (C++ function), 1367
esp_aptrace_fseek (C++ function), 1367
esp_aptrace_fstop (C++ function), 1367
esp_aptrace_ftell (C++ function), 1367
esp_aptrace_fwrite (C++ function), 1367
esp_aptrace_host_is_connected (C++ function), 1366
esp_aptrace_init (C++ function), 1364
esp_aptrace_read (C++ function), 1366
esp_aptrace_vprintf (C++ function), 1365
esp_aptrace_vprintf_to (C++ function), 1365
esp_aptrace_write (C++ function), 1365
esp_async_memcpy (C++ function), 1728
esp_async_memcpy_install (C++ function), 1728
esp_async_memcpy_install_cpudma (C++ function), 1727
esp_async_memcpy_uninstall (C++ function), 1728
esp_base_mac_addr_get (C++ function), 1651
esp_base_mac_addr_set (C++ function), 1650
ESP_BOOTLOADER_DESC_MAGIC_BYTE (C macro), 1363
esp_bootloader_desc_t (C++ struct), 1363
esp_bootloader_desc_t::date_time (C++ member), 1363
esp_bootloader_desc_t::idf_ver (C++ member), 1363
esp_bootloader_desc_t::magic_byte (C++ member), 1363
esp_bootloader_desc_t::reserved (C++ member), 1363
esp_bootloader_desc_t::reserved2 (C++ member), 1363
esp_bootloader_get_description (C++ function), 1363
esp_btbb_disable (C++ function), 1990
esp_btbb_enable (C++ function), 1990
esp_cache_msync (C++ function), 1604
ESP_CACHE_MSYNC_FLAG_DIR_C2M (C macro), 1605
ESP_CACHE_MSYNC_FLAG_DIR_M2C (C macro), 1606
ESP_CACHE_MSYNC_FLAG_INVALIDATE (C macro), 1605
ESP_CACHE_MSYNC_FLAG_TYPE_DATA (C macro), 1606
ESP_CACHE_MSYNC_FLAG_TYPE_INST (C macro), 1606
ESP_CACHE_MSYNC_FLAG_UNALIGNED (C macro), 1605
esp_chip_id_t (C++ enum), 1359
esp_chip_id_t::ESP_CHIP_ID_ESP32 (C++ enumerator), 1359
esp_chip_id_t::ESP_CHIP_ID_ESP32C2 (C++ enumerator), 1359
esp_chip_id_t::ESP_CHIP_ID_ESP32C3 (C++ enumerator), 1359
esp_chip_id_t::ESP_CHIP_ID_ESP32C6 (C++ enumerator), 1360
esp_chip_id_t::ESP_CHIP_ID_ESP32H2 (C++ enumerator), 1360
esp_chip_id_t::ESP_CHIP_ID_ESP32P4 (C++ enumerator), 1360
esp_chip_id_t::ESP_CHIP_ID_ESP32S2 (C++ enumerator), 1359
esp_chip_id_t::ESP_CHIP_ID_ESP32S3 (C++ enumerator), 1359
esp_chip_id_t::ESP_CHIP_ID_INVALID (C++ enumerator), 1360
esp_chip_info (C++ function), 1653
esp_chip_info_t (C++ struct), 1653
esp_chip_info_t::cores (C++ member), 1653
esp_chip_info_t::features (C++ member), 1653
esp_chip_info_t::model (C++ member), 1653
esp_chip_info_t::revision (C++ member), 1653

Espressif Systems 2222 Release master

Submit Document Feedback
Index

ESP_DRAM_LOGE (C macro), 1643
ESP_DRAM_LOGI (C macro), 1643
ESP_DRAM_LOGV (C macro), 1643
ESP_DRAM_LOGW (C macro), 1643
ESP_DS_C_LEN (C macro), 541
esp_ds_context_t (C++ type), 541
esp_ds_data_t (C++ type), 541
esp_ds_encrypt_params (C++ function), 539
esp_ds_finish_sign (C++ function), 539
esp_ds_is_busy (C++ function), 539
ESP_DS_IV_BIT_LEN (C macro), 541
ESP_DS_IV_LEN (C macro), 541
dsp_p_data_t (C++ struct), 540
dsp_p_data_t::length (C++ member), 540
dsp_p_data_t::M (C++ member), 540
dsp_p_data_t::M_prime (C++ member), 540
dsp_p_data_t::Rb (C++ member), 540
dsp_p_data_t::Y (C++ member), 540
dsp_p_data_t::Y (C++ member), 540
ESP_DS_SIG.med handling (C macro), 541
ESP_DS_SIGNATURE_L_BIT_LEN (C macro), 541
ESP_DS_SIGNATURE_M_BIT_LEN (C macro), 541
ESP_DS_SIGNATURE_MD_BIT_LEN (C macro), 541
ESP_DS_SIGNATURE_PADDING_BIT_LEN (C macro), 541
dsp_start_sign (C++ function), 538
dsp_eap_client_clear_ca_cert (C++ function), 339
dsp_eap_client_clear_certificate_and_key (C++ function), 339
dsp_eap_client_clear_identity (C++ function), 338
dsp_eap_client_clear_new_password (C++ function), 338
dsp_eap_client_clear_password (C++ function), 338
dsp_eap_client_clear_username (C++ function), 338
dsp_eap_client_get_disable_time_check (C++ function), 340
dsp_eap_client_get_disable_time_check (C++ function), 340
dsp_eap_client_get_disable_time_check (C++ function), 340
dsp_eap_client_set_fast_params (C++ function), 340
dsp_eap_client_set_password (C++ function), 338
dsp_eap_client_set_suiteb_192bit_certification (C++ function), 340
dsp_eap_client_set_ttls_phase2_method (C++ function), 340
dsp_eap_client_set_username (C++ function), 338
dsp_eap_client_use_default_cert_bundle (C++ function), 340
dsp_eap_fast_config (C++ struct), 341
dsp_eap_fast_config::fast_max_pac_list_len (C++ member), 341
dsp_eap_fast_config::fast_pac_format_binary (C++ member), 341
dsp_eap_fast_config::fast_provisioning (C++ member), 341
dsp_eap_ttls_phase2_types (C++ enum), 341
dsp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_CHAP (C++ enumerator), 341
dsp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_EAP (C++ enumerator), 341
dsp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_MSCHAP (C++ enumerator), 341
dsp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_MSCHAPV2 (C++ enumerator), 341
dsp_eap_ttls_phase2_types::ESP_EAP_TTLS_PHASE2_PAP (C++ enumerator), 341
ESP_EARLY_LOGD (C macro), 1642
ESP_EARLY_LOGE (C macro), 1642
ESP_EARLY_LOGI (C macro), 1642
ESP_EARLY_LOGV (C macro), 1642
ESP_EARLY_LOGW (C macro), 1642
dsp_eufuse_batch_write_begin (C++ function), 1407
dsp_eufuse_batch_write_cancel (C++ function), 1408
dsp_eufuse_batch_write_commit (C++ function), 1408
dsp_eufuse_block_is_empty (C++ function), 1408
dsp_eufuse_block_t (C++ enum), 1399
dsp_eufuse_block_t::EFUSE_BLK0 (C++ enumerator), 1399
dsp_eufuse_block_t::EFUSE_BLK1 (C++ enumerator), 1399
dsp_eufuse_block_t::EFUSE_BLK10 (C++ enumerator), 1400
dsp_eufuse_block_t::EFUSE_BLK1 (C++ enumerator), 1399
dsp_eufuse_block_t::EFUSE_BLK2 (C++ enumerator), 1399
dsp_eufuse_block_t::EFUSE_BLK3 (C++ enumerator), 1400
dsp_eufuse_block_t::EFUSE_BLK4 (C++ enumerator), 1400
dsp_eufuse_block_t::EFUSE_BLK5 (C++ enumerator), 1400
dsp_eufuse_block_t::EFUSE_BLK6 (C++ enumerator), 1400

Espressif Systems 2225 Release master
Submit Document Feedback
esp_efuse_block_t::EFUSE_BLK7 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK8 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK9 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY0 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY1 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY2 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY3 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY4 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY5 (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_KEY_MAX (C++ enumerator), 1400
esp_efuse_block_t::EFUSE_BLK_MAX (C++ enumerator), 1401
esp_efuse_block_t::EFUSE_BLK_SYS_DATA (C++ enumerator), 1399
esp_efuse_block_t::EFUSE_BLK_SYS_DATA_PART2 (C++ enumerator), 1401
esp_efuse_block_t::EFUSE_BLK_SYS_DATA_PART1 (C++ enumerator), 1401
esp_efuse_check_errors (C++ function), 1412
esp_efuse_check_secure_version (C++ function), 1407
esp_efuse_coding_scheme_t::ENUM (C++ enum), 1401
esp_efuse_coding_scheme_t::ENUM (C++ enum), 1401
esp_efuse_coding_scheme_t::ENUM (C++ enum), 1401
esp_efuse_count_unused_key_blocks (C++ function), 1410
esp_efuse_desc_t::EFUSE_BLK7 (C++ struct), 1412
esp_efuse_desc_t::bit_count (C++ member), 1412
esp_efuse_desc_t::bit_start (C++ member), 1412
esp_efuse_desc_t::efuse_block (C++ member), 1412
esp_efuse_disable_rom_download_mode (C++ function), 1406
esp_efuse_enable_rom_secure_download_mode (C++ function), 1406
esp_efuse_find_purpose (C++ function), 1409
esp_efuse_find_unused_key_block (C++ function), 1410
esp_efuse_get_coding_scheme (C++ function), 1405
esp_efuse_get_digest_revoke (C++ function), 1410
esp_efuse_get_field_size (C++ function), 1404
esp_efuse_get_key (C++ function), 1410
esp_efuse_get_key_dis_read (C++ function), 1408
esp_efuse_get_key_dis_write (C++ function), 1408
esp_efuse_get_key_purpose (C++ function), 1409
esp_efuse_get_key_purpose_dis_write (C++ function), 1409
esp_efuse_get_purpose_field (C++ function), 1405
esp_efuse_get_write_protect_of_digest_revoke (C++ function), 1410
esp_efuse_key_block_unused (C++ function), 1409
esp_efuse_mac_get_custom (C++ function), 1651
esp_efuse_mac_get_default (C++ function), 1651
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_MAX (C++ function), 1409
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_UP (C++ function), 1401
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG (C++ function), 1401
esp_efuse_purpose_t::ESP_EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE (C++ function), 1401
esp_efuse_read_secure_version (C++ function), 1401
esp_efuse_read_reg (C++ function), 1404
esp_efuse_read_secure_version (C++ function), 1407
ESP_ERR_HTTP_EAGAIN (C macro), 142
ESP_ERR_HTTP_FETCH_HEADER (C macro), 142
ESP_ERR_HTTP_INVALID_TRANSPORT (C macro), 142
ESP_ERR_HTTP_MAX_REDIRECT (C macro), 142
ESP_ERR_HTTP_WRITE_DATA (C macro), 142
ESP_ERR_HTTPD_ALLOC_MEM (C macro), 197
ESP_ERR_HTTPD_BASE (C macro), 197
ESP_ERR_HTTPD_HANDLER_EXISTS (C macro), 197
ESP_ERR_HTTPD_HANDLERS_FULL (C macro), 197
ESP_ERR_HTTPD_INVALID_REQ (C macro), 197
ESP_ERR_HTTPD_RESP_HDR (C macro), 197
ESP_ERR_HTTPD_RESP_SEND (C macro), 197
ESP_ERR_HTTPD_RESULT_TRUNC (C macro), 197
ESP_ERR_HTTPD_TASK (C macro), 197
ESP_ERR_HTTPS_OTA_BASE (C macro), 1422
ESP_ERR_HTTPS_OTA_IN_PROGRESS (C macro), 1422
ESP_ERR_HW_CRYPTO_BASE (C macro), 1416
ESP_ERR_INVALID_ARG (C macro), 1415
ESP_ERR_INVALID_CRC (C macro), 1415
ESP_ERR_INVALID_MAC (C macro), 1415
ESP_ERR_INVALID_RESPONSE (C macro), 1415
ESP_ERR_INVALID_SIZE (C macro), 1415
ESP_ERR_INVALID_STATE (C macro), 1415
ESP_ERR_INVALID_VERSION (C macro), 1415
ESP_ERR_MBEDTLS_CERT_PARTLY_OK (C macro), 126
ESP_ERR_MBEDTLS_CTR_DRBG_SEED_FAILED (C macro), 126
ESP_ERR_MBEDTLS_PK_PARSE_KEY_FAILED (C macro), 127
ESP_ERR_MBEDTLS_SSL_CONF_ALPN_PROTOCOL (C macro), 126
ESP_ERR_MBEDTLS_SSL_CONF_OWNER_CERT_FAILED (C macro), 127
ESP_ERR_MBEDTLS_SSL_CONF_PSK_FAILED (C macro), 127
ESP_ERR_MBEDTLS_SSL_CONFIG_DEFAULTS_FAILED (C macro), 126
ESP_ERR_MBEDTLS_SSL_HANDSHAKE_FAILED (C macro), 127
ESP_ERR_MBEDTLS_SSL_SET_HOSTNAME_FAILED (C macro), 126
ESP_ERR_MBEDTLS_SSL_SETUP_FAILED (C macro), 127
ESP_ERR_MBEDTLS_SSL_TICKET_SETUP_FAILED (C macro), 127
ESP_ERR_MBEDTLS_SSL_WRITE_FAILED (C macro), 127
ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED (C macro), 127
ESP_ERR_MEMPROT_BASE (C macro), 1416
ESP_ERR_MESH_ARGUMENT (C macro), 260
ESP_ERR_MESH_BASE (C macro), 1416
ESP_ERR_MESH_DISCARD (C macro), 261
ESP_ERR_MESH_DISCARD_DUPLICATE (C macro), 260
ESP_ERR_MESH_DISCONNECTED (C macro), 260
ESP_ERR_MESH_EXCEED_MTU (C macro), 260
ESP_ERR_MESH_INTERFACE (C macro), 260
ESP_ERR_MESH_NO_MEMORY (C macro), 260
ESP_ERR_MESH_NO_PARENT_FOUND (C macro), 260
ESP_ERR_MESH_NO_ROUTE_FOUND (C macro), 260
ESP_ERR_MESH_NOT_ALLOWED (C macro), 260
ESP_ERR_MESH_NOT_CONFIG (C macro), 259
ESP_ERR_MESH_NOT_INIT (C macro), 259
ESP_ERR_MESH_NOT_START (C macro), 260
ESP_ERR_MESH_OPTION_NULL (C macro), 260
ESP_ERR_MESH_OPTION_UNKNOWN (C macro), 260
ESP_ERR_MESH_PS (C macro), 261
ESP_ERR_MESH_QUEUE_FAIL (C macro), 260
ESP_ERR_MESH_QUEUE_FULL (C macro), 260
ESP_ERR_MESH_QUEUE_READ (C macro), 261
ESP_ERR_MESH_RECV_RELEASE (C macro), 261
ESP_ERR_MESH_TIMEOUT (C macro), 260
ESP_ERR_MESH_VOTING (C macro), 261
ESP_ERR_MESH_WIFI_NOT_START (C macro), 259
ESP_ERR_MESH_XMIT (C macro), 261
ESP_ERR_MESH_XON_NO_WINDOW (C macro), 260
ESP_ERR_NO_MEM (C macro), 1415
ESP_ERR_NOT_ALLOWED (C macro), 1416
ESP_ERR_NOT_ENOUGH_UNUSED_KEY_BLOCKS (C macro), 1413
ESP_ERR_NOT_FINISHED (C macro), 1416
ESP_ERR_NOT_SUPPORTED (C macro), 1415
ESP_ERR_NVS_BASE (C macro), 1300
ESP_ERR_NVS_CONTENT_DIFERS (C macro), 1302
ESP_ERR_NVS_CORRUPT_KEY_PART (C macro), 1301
ESP_ERR_NVS_DEFAULTS_FAILED (C macro), 1301
ESP_ERR_NVS_ENCR_NOT_SUPPORTED (C macro), 1301
ESP_ERR_NVS_INVALID_HANDLE (C macro), 1300
ESP_ERR_NVS_INVALID_LENGTH (C macro), 1301
ESP_ERR_NVS_INVALID_NAME (C macro), 1300
ESP_ERR_NVS_INVALID_STATE (C macro), 1301
ESP_ERR_NVS_KEY_TOO_LONG (C macro), 1301
ESP_ERR_NVS_KEYS_NOT_INITIALIZED (C macro), 1301
ESP_ERR_NVS_NEW_VERSION_FOUND (C macro), 1301
ESP_ERR_NVS_NOT_ALLOWED (C macro), 1301
ESP_ERR_NVS_NOT_ENOUGH_SPACE (C macro), 1300
ESP_ERR_NVS_NOT_FOUND (C macro), 1300
ESP_ERR_NVS_NOTInicialized (C macro), 1300
ESP_ERR_NVS_NO_FREE_PAGES (C macro), 1301
ESP_ERR_NVS_NO_INIT (C macro), 1300
ESP_ERR_NVS_NOT_INITIALIZED (C macro), 1300
ESP_ERR_NVS_NO_MEM (C macro), 1300
ESP_ERR_NOT_ALLOWED (C macro), 1301
ESP_ERR_NO_MEM (C macro), 1301
ESP_ERR_NVS_ENCR_NOT_SUPPORTED (C macro), 1301
ESP_ERR_NVS_INVALID_STATE (C macro), 1301
ESP_ERR_NVS_KEY_TOO_LONG (C macro), 1301
ESP_ERR_NVS_KEYS_NOT_INITIALIZED (C macro), 1301
ESP_ERR_NVS_NEW_VERSION_FOUND (C macro), 1301
ESP_ERR_NVS_NOT_ALLOWED (C macro), 1301
ESP_ERR_NVS_NOT_ENOUGH_SPACE (C macro), 1300
ESP_ERR_NVS_NOT_FOUND (C macro), 1300
ESP_ERR_NVS_NOT_INITIALIZED (C macro), 1300
ESP_ERR_MBEDTLS_X509_CRT_PARSE_FAILED (C macro), 127
ESP_ERR_MEMPROT_BASE (C macro), 1416
ESP_ERR_MESH_ARGUMENT (C macro), 260
ESP_ERR_MESH_BASE (C macro), 1416
ESP_ERR_MESH_DISCARD (C macro), 261

Index
ESP_ERR_NVS_PAGE_FULL (C macro), 1301
ESP_ERR_NVS_PART_NOT_FOUND (C macro), 1301
ESP_ERR_NVS_READ_ONLY (C macro), 1300
ESP_ERR_NVS_REMOVE_FAILED (C macro), 1300
ESP_ERR_NVS_SEC_BASE (C macro), 1309
ESP_ERR_NVS_SEC_HMAC_KEY_BLK_ALREADY_UNUSED (C macro), 1309
ESP_ERR_NVS_SEC_HMAC_KEY_GENERATION_FAILURE (C macro), 1309
ESP_ERR_NVS_SEC_HMAC_KEY_NOT_FOUND (C macro), 1309
ESP_ERR_NVS_SEC_HMAC_XTS_KEYS_DERIV_FAILURE (C macro), 1309
ESP_ERR_NVS_TYPE_MISMATCH (C macro), 1300
ESP_ERR_NVS_VALUE_TOO_LONG (C macro), 1301
ESP_ERR_NVS_WRONG_ENCRYPTION (C macro), 1301
ESP_ERR_NVS_XTS_CFG_FAILED (C macro), 1301
ESP_ERR_NVS_XTS_CFG_NOT_FOUND (C macro), 1301
ESP_ERR_NVS_XTS_DECR_FAILED (C macro), 1301
ESP_ERR_NVS_XTS_ENCR_FAILED (C macro), 1301
ESP_ERR_OTA_BASE (C macro), 1672
ESP_ERR_OTA_PARTITION_CONFLICT (C macro), 1672
ESP_ERR_OTA_ROLLBACK_FAILED (C macro), 1672
ESP_ERR_OTA_ROLLBACK_INVALID_STATE (C macro), 1672
ESP_ERR_OTA_SELECT_INFO_INVALID (C macro), 1672
ESP_ERR_OTA_SMALL_SEC_VER (C macro), 1672
ESP_ERR_OTA_VALIDATE_FAILED (C macro), 1672
esp_err_t (C++ type), 1416
ESP_ERR_TIMEOUT (C macro), 1415
esp_err_to_name (C++ function), 1414
esp_err_to_name_t (C++ function), 1414
ESP_ERR_ULP_BASE (C macro), 1769
ESP_ERR_ULP_BRANCH_OUT_OF_RANGE (C macro), 1770
ESP_ERR_ULP_DUPLICATE_LABEL (C macro), 1770
ESP_ERR_ULP_INVALID_LOAD_ADDR (C macro), 1770
ESP_ERR_ULP_SIZE_TOO_BIG (C macro), 1769
ESP_ERR_ULP_UNDEFINED_LABEL (C macro), 1770
ESP_ERR_WIFI_BASE (C macro), 1416
ESP_ERR_WIFI_CONN (C macro), 291
ESP_ERR_WIFI_DISCARD (C macro), 292
ESP_ERR_WIFI_IP (C macro), 291
ESP_ERR_WIFI_INIT_STATE (C macro), 291
ESP_ERR_WIFI_MAC (C macro), 291
ESP_ERR_WIFI_MODE (C macro), 291
ESP_ERR_WIFI_NOT_ASSOC (C macro), 291
ESP_ERR_WIFI_NOT_CONNECT (C macro), 291
ESP_ERR_WIFI_NOT_INIT (C macro), 290
ESP_ERR_WIFI_NOT_STARTED (C macro), 290
ESP_ERR_WIFI_NOT_STOPPED (C macro), 290
ESP_ERR_WIFI_NOT_NVS (C macro), 291
ESP_ERR_WIFI_PASSWORD (C macro), 291
ESP_ERR_WIFI_POST (C macro), 291
ESP_ERR_WIFI_REGISTRAR (C macro), 344
ESP_ERR_WIFI_STATE (C macro), 291
ESP_ERR_WIFI_STOP_STATE (C macro), 291
ESP_ERR_WIFI_TIMEOUT (C macro), 291
ESP_ERR_WIFI_TWT_FULL (C macro), 292
ESP_ERR_WIFI_TWT_SETUP_REJECT (C macro), 292
ESP_ERR_WIFI_TWT_SETUP_TIMEOUT (C macro), 292
ESP_ERR_WIFI_TWT_SETUP_TXFAIL (C macro), 292
ESP_ERR_WIFI_TX_DISALLOW (C macro), 292
ESP_ERR_WIFI_WAKE_FAIL (C macro), 291
ESP_ERR_WIFI_WOULD_BLOCK (C macro), 291
ESP_ERR_WIFI_WPS_SM (C macro), 344
ESP_ERR_WIFI_WPS_TYPE (C macro), 344
ESP_ERR_WOLFSSSL_CERT_VERIFY_SETUP_FAILED (C macro), 127
ESP_ERR_WOLFSSSL_CTX_SETUP_FAILED (C macro), 127
ESP_ERR_WOLFSSSL_KEY_VERIFY_SETUP_FAILED (C macro), 127
ESP_ERR_WOLFSSSL_SSL_CONF_ALPN_PROTOCOLS_FAILED (C macro), 127
ESP_ERR_WOLFSSSL_SSL_HANDSHAKE_FAILED (C macro), 127
ESP_ERR_WOLFSSSL_SSL_SET_HOSTNAME_FAILED (C macro), 127
ESP_ERR_WOLFSSSL_SSL_SET_INVALID (C macro), 127
ESP_ERR_WOLFSSSL_SSL_WRITE_FAILED (C macro), 127
ESP_ERROR_CHECK (C macro), 1416
ESP_ERROR_CHECK WITHOUT_ABORT (C macro), 1416
esp_eth_config_t::stack_input (C macro), 268
esp_eth_config_t::read_phy_reg (C macro), 291
esp_eth_config_t::on_lowlevel_init_done (C macro), 291
esp_eth_config_t::on_lowlevel_deinit_done (C macro), 291
esp_eth_config_t::esp_esptouch_set_timeout (C macro), 291
esp_eth_config_t::esp_esptouch_set_timeout (C++ function), 291
esp_eth_config_t::esp_esptouch_set_timeout (C++ member), 291
esp_eth_config_t::esp_esptouch_set_timeout (C++ struct), 291
esp_eth_config_t::check_link_period_ms (C++ function), 291
esp_eth_config_t::check_link_period_ms (C++ member), 291
esp_eth_config_t::mac (C++ member), 291
esp_eth_config_t::on_lowlevel_deinit_done (C++ member), 291
esp_eth_config_t::on_lowlevel_init_done (C++ member), 291
esp_eth_config_t::phy (C++ member), 291
esp_eth_config_t::read_phy_reg (C++ member), 291
esp_eth_config_t::stack_input (C++ member), 291

Espressif Systems 2229 Release master
Submit Document Feedback
esp_eth_config_t::write_phy_reg (C++ member), 368
esp_eth_config_t::write_phy_reg (C++ function), 366
esp_eth_del_netif_glue (C++ function), 389
esp_eth_driver_install (C++ function), 363
esp_eth_driver_uninstall (C++ function), 364
esp_eth_handle_t (C++ type), 368
esp_eth_increase_reference (C++ function), 366
esp_eth_ioctl (C++ function), 369
esp_eth_ioctl_t (C++ enumerator), 369
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ type), 383
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ struct), 383
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ function), 384
esp_eth_ioctl_t::ETH_CMD_WRITE_PHY_REG (C++ member), 385
esp_eth_phy_802_3_read_oui (C++ function), 388
esp_eth_phy_802_3_reset (C++ function), 385
esp_eth_phy_802_3_reset_hw (C++ function), 387
esp_eth_phy_802_3_set_addr (C++ function), 386
esp_eth_phy_802_3_set_duplex (C++ function), 386
esp_eth_phy_802_3_set_mediator (C++ function), 385
esp_eth_phy_802_3_set_speed (C++ function), 386
ESP_ETH_PHY_ADDR_AUTO (C macro), 384
esp_eth_phy_into_phy_802_3 (C++ function), 388
esp_eth_phy_new_dp83848 (C++ function), 380
esp_eth_phy_new_ip101 (C++ function), 380
esp_eth_phy_new_ksz80xx (C++ function), 380
esp_eth_phy_new_lan87xx (C++ function), 380
esp_eth_phy_new_rtI8201 (C++ function), 380
esp_eth_phy_reg_rw_data_t::reg_addr (C++ member), 368
esp_eth_phy_reg_rw_data_t::reg_value_p (C++ member), 368
esp_eth_phy_s::advertise_pause_ability (C++ member), 382
esp_eth_phy_s::autonego Ctrl (C++ member), 382
esp_eth_phy_s::custom_ioctl (C++ member), 383
esp_eth_phy_s::deinit (C++ member), 381
esp_eth_phy_s::del (C++ member), 384
esp_eth_phy_s::get_addr (C++ member), 382
esp_eth_phy_s::get_link (C++ member), 382
esp_eth_phy_s::init (C++ member), 381
esp_eth_phy_s::loopback (C++ member), 383
esp_eth_phy_s::pwrctl (C++ member), 382
esp_eth_phy_s::reset (C++ member), 381
esp_eth_phy_s::reset_hw (C++ member), 381
esp_eth_phy_s::set_addr (C++ member), 382
esp_eth_phy_s::set_duplex (C++ member), 383
esp_eth_phy_s::set_mediator (C++ member), 381
esp_eth_phy_s::set_speed (C++ member), 383
esp_eth_phy_t (C++ type), 384
esp_eth_start (C++ function), 364
esp_eth_state_t (C++ enum), 371
esp_eth_state_t::ETH_STATE_DEINIT (C++ enumerator), 371
esp_eth_state_t::ETH_STATE_DUPELEX (C++ enumerator), 371
esp_eth_state_t::ETH_STATE_LINK (C++ enumerator), 371
esp_eth_state_t::ETH_STATE_LLINIT (C++ enumerator), 371
esp_eth_state_t::ETH_STATE_PAUSE (C++ enumerator), 371
esp_eth_state_t::ETH_STATE_SPEED (C++ enumerator), 371
esp_eth_stop (C++ function), 364
esp_eth_transmit (C++ function), 365
esp_eth_transmit_vargs (C++ function), 365
esp_eth_update_input_path (C++ function), 364
ESP_EVENT_ANY_BASE (C macro), 1436
ESP_EVENT_ANY_ID (C macro), 1436
ESP_EVENT_DECLARE_BASE (C macro), 1436
ESP_EVENT_DEFINE_BASE (C macro), 1436
esp_event_dump (C++ function), 1434
esp_event_handler_instance_register (C++ function), 1431
esp_event_handler_instance_register_with (C++ function), 1430
esp_event_handler_instance_t (C++ type), 1436
esp_event_handler_instance_unregister (C++ function), 1432
esp_event_handler_unregister_with (C++ function), 1432
esp_event_handler_register (C++ function), 1429
esp_event_handler_register_with (C++ function), 1429
esp_event_handler_t (C++ type), 1436
esp_event_handler_unregister (C++ function), 1431
esp_event_handler_unregister_with (C++ function), 1432
esp_event_isr_post (C++ function), 1433
esp_event_isr_post_to (C++ function), 1434
esp_event_loop_args_t::task_stack_size (C++ member), 1435
esp_event_loop_args_t::task_priority (C++ member), 1435
esp_event_loop_args_t::task_name (C++ member), 1436
esp_event_loop_args_t::task_core_id (C++ member), 1436
esp_event_loop_args_t::task_stack_size (C++ member), 1435
esp_event_loop_create (C++ function), 1428
esp_event_loop_create_default (C++ function), 1428
esp_event_loop_delete (C++ function), 1428
esp_event_loop_delete_default (C++ function), 1428
esp_event_loop_handler_t (C++ type), 1436
esp_event_loop_run (C++ function), 1428
esp_event_post (C++ function), 1433
esp_event_post_to (C++ function), 1433
ESP_EXECUTE_EXPRESSION_WITH_STACK
    (C macro), 1370
esp_execute_shared_stack_function
    (C++ function), 1370
ESP.FAIL (C macro), 1415
esp_fill_random (C++ function), 1689
esp_flash_chip_driver_initialized
    (C++ function), 691
esp_flash_counter_t (C++ struct), 708
esp_flash_counter_t::bytes (C++ member), 708
esp_flash_counter_t::count (C++ member), 708
esp_flash_counter_t::time (C++ member), 708
esp_flash_counters_t (C++ struct), 708
esp_flash_counters_t::erase (C++ member), 708
esp_flash_counters_t::read (C++ member), 708
esp_flash_counters_t::write (C++ member), 708
esp_flash_dump_counters (C++ function), 707
esp_flash_enc_mode_t (C++ enum), 711
esp_flash_enc_mode_t::ESP_FLASH_ENC_MODE_RELEASE
    (C++ enumerator), 711
esp_flash_enc_mode_t::ESP_FLASH_ENC_MODE_DISABLED
    (C++ enum), 708
esp_flash_enc_mode_t::ESP_FLASH_ENC_MODE_DEVELOPMENT
    (C++ enum), 708
esp_flash_encryption_check_and_update
    (C++ function), 709
esp_flash_encrypt_check_and_update
    (C++ function), 709
esp_flash_encrypt_enable (C++ function), 709
esp_flash_encrypt_init (C++ function), 709
esp_flash_encrypt_initiated_once
    (C++ function), 709
esp_flash_encrypt_is_write_protected
    (C++ function), 709
esp_flash_encrypt_region (C++ function), 709
esp_flash_encrypt_state (C++ function), 709
esp_flash_encryption_cfg_verify_release_mode
    (C++ function), 710
esp_flash_encryption_enable_secure_features
    (C++ function), 709
esp_flash_encryption_enabled (C++ function), 709
esp_flash_encryption_init_checks (C++ function), 710
esp_flash_encryption_set_release_mode
    (C++ function), 710
esp_flash_erase_chip (C++ function), 692
esp_flash_erase_region (C++ function), 692
esp_flash_get_chip_write_protect (C++ function), 692
esp_flash_get_counters (C++ function), 708
esp_flash_get_physical_size (C++ function), 691
esp_flash_get_protectable_regions
    (C++ function), 693
esp_flash_get_protected_region (C++ function), 693
esp_flash_get_region (C++ function), 691
esp_flash_init (C++ function), 690
esp_flash_io_size (C++ function), 706
esp_flash_io_mode_t (C++ enum), 706
esp_flash_io_mode_t::SPI_FLASH_DOUT
    (C++ enumerator), 706
esp_flash_io_mode_t::SPI_FLASH_DOUT
    (C++ function), 691
esp_flash_io_mode_t::SPI_FLASH_FASTRD
    (C++ function), 706
esp_flash_io_mode_t::SPI_FLASH_OPI_DTR
    (C++ function), 691
esp_flash_io_mode_t::SPI_FLASH_OPI_STR
    (C++ function), 706
esp_flash_io_mode_t::SPI_FLASH_QIO
    (C++ function), 706
esp_flash_io_mode_t::SPI_FLASH_QOUT
    (C++ function), 691
esp_flash_is_quad_mode (C++ function), 693
esp_flash_os_functions_t (C++ struct), 696
esp_flash_os_functions_t::check_yield
    (C++ member), 696
esp_flash_os_functions_t::delay_us
    (C++ member), 696
esp_flash_os_functions_t::end
    (C++ member), 696
esp_flash_os_functions_t::get_system_time
    (C++ member), 696
esp_flash_os_functions_t::get_temp_buffer
    (C++ member), 696
esp_flash_os_functions_t::region_protected
    (C++ member), 696
esp_flash_os_functions_t::release_temp_buffer
    (C++ member), 697
esp_flash_os_functions_t::set_flash_op_status
    (C++ member), 697
esp_flash_os_functions_t::start
    (C++ member), 696
esp_flash_read (C++ function), 694
esp_flash_read_encrypted (C++ function), 695
esp_flash_read_id (C++ function), 691
esp_flash_read_unique_chip_id (C++ function), 691
esp_flash_region_t (C++ struct), 696
esp_flash_region_t::offset (C++ member),
esp_flash_region_t::size (C++ member), 696
esp_flash_region_t (C++ member), 696
esp_flash_reset_counters (C++ function), 707
esp_flash_set_chip_write_protect (C++ function), 693
esp_flash_set_protected_region (C++ function), 694
esp_flash_speed_s::ESP_FLASH_10MHZ (C++ enumerator), 705
esp_flash_speed_s::ESP_FLASH_120MHZ (C++ enumerator), 706
esp_flash_speed_s::ESP_FLASH_20MHZ (C++ enumerator), 705
esp_flash_speed_s::ESP_FLASH_26MHZ (C++ enumerator), 705
esp_flash_speed_s::ESP_FLASH_40MHZ (C++ enumerator), 706
esp_flash_speed_s::ESP_FLASH_5MHZ (C++ enumerator), 705
esp_flash_speed_s::ESP_FLASH_80MHZ (C++ enumerator), 706
esp_flash_speed_t::ESP_FLASH_SPEED_MAX (C++ enumerator), 706
esp_flash_speed_t (C++ type), 705
esp_flash_spi_device_config_t (C++ struct), 689
esp_flash_spi_device_config_t::cs_id (C++ member), 690
esp_flash_spi_device_config_t::cs_io_num (C++ member), 690
esp_flash_spi_device_config_t::freq_mhz (C++ member), 690
esp_flash_spi_device_config_t::host_id (C++ member), 689
esp_flash_spi_device_config_t::input_delay_ns (C++ member), 690
esp_flash_spi_device_config_t::io_mode (C++ member), 690
esp_flash_spi_device_config_t::speed (C++ member), 690
esp_flash_t (C++ struct), 697
esp_flash_t::busy (C++ member), 697
esp_flash_t::chip_drv (C++ member), 697
esp_flash_t::chip_id (C++ member), 697
esp_flash_t::host (C++ member), 697
esp_flash_t::hp_dummy_ena (C++ member), 697
esp_flash_t::os_func (C++ member), 697
esp_flash_t::os_func_data (C++ member), 697
esp_flash_t::read_mode (C++ member), 697
esp_flash_t::reserved_flags (C++ member), 697
esp_flash_t::size (C++ member), 697
esp_flash_write (C++ function), 694
esp_flash_write_encrypted (C++ function), 695
esp_flash_write_protect_crypt_cnt (C++ function), 710
esp_freesRTOS_idcl_cb_t (C++ type), 1575
esp_freesRTOS_tick_cb_t (C++ type), 1575
gcov_dump (C++ function), 1367
esp_get_deep_sleep_wake_stub (C++ function), 1702
esp_get_flash_encryption_mode (C++ function), 710
gesp_get_free_heap_size (C++ function), 1648
esp_get_free_internal_heap_size (C++ function), 1648
esp_get_idf_version (C++ function), 1649
gesp_get_minimum_free_heap_size (C++ function), 1648
ESP_GOTO_ON_ERROR (C macro), 1414
ESP_GOTO_ON_ERROR_ISR (C macro), 1414
ESP_GOTO_ON_FALSE (C macro), 1414
esp_hmac_calculate (C++ function), 534
gesp_hmac_enable (C++ function), 534
gesp_hmac_disable (C++ function), 535
gesp_http_client_add_auth (C++ function), 137
gesp_http_client_auth_type_t (C++ enum), 145
gesp_http_client_auth_type_t::HTTP_AUTH_TYPE_BASIC (C++ enum), 145
gesp_http_client_auth_type_t::HTTP_AUTH_TYPE_DIGEST (C++ enum), 145
gesp_http_client_auth_type_t::HTTP_AUTH_TYPE_NONE (C++ enum), 145
gesp_http_client_cancel_request (C++ function), 132
gesp_http_client_cleanup (C++ function), 136
gesp_http_client_close (C++ function), 136
gesp_http_client_config_t (C++ struct), 139
gesp_http_client_config_t::auth_type (C++ member), 140
gesp_http_client_config_t::buffer_size (C++ member), 141
gesp_http_client_config_t::buffer_size_tx (C++ member), 141
gesp_http_client_config_t::cert_len (C++ member), 140
gesp_http_client_config_t::cert_pem (C++ member), 140
gesp_http_client_config_t::client_cert_len (C++ member), 140
gesp_http_client_config_t::client_cert_pem (C++ member), 140
gesp_http_client_config_t::client_key_len (C++ member), 140
gesp_http_client_config_t::client_key_password (C++ member), 140
gesp_http_client_config_t::client_key_password_len (C++ member), 140
(C++ member), 140 esp_http_client_config_t::client_key_pem (C++ member), 140 esp_http_client_config_t::common_name (C++ member), 141 esp_http_client_config_t::crt_bundle_attach (C++ member), 141 esp_http_client_config_t::disable_auto_redirect (C++ member), 140 esp_http_client_config_t::ds_data (C++ member), 142 esp_http_client_config_t::event_handler (C++ member), 141 esp_http_client_config_t::host (C++ member), 139 esp_http_client_config_t::if_name (C++ member), 142 esp_http_client_config_t::is_async (C++ member), 141 esp_http_client_config_t::keep_alive_count (C++ member), 142 esp_http_client_config_t::keep_alive_enable (C++ member), 141 esp_http_client_config_t::keep_alive_idle (C++ member), 141 esp_http_client_config_t::keep_alive_interval (C++ member), 141 esp_http_client_config_t::max_authorization_redirects (C++ member), 141 esp_http_client_config_t::max_redirection_count (C++ member), 141 esp_http_client_config_t::method (C++ member), 140 esp_http_client_config_t::password (C++ member), 139 esp_http_client_config_t::path (C++ member), 140 esp_http_client_config_t::port (C++ member), 139 esp_http_client_config_t::query (C++ member), 140 esp_http_client_config_t::skip_cert_common_name_check (C++ member), 141 esp_http_client_config_t::timeout_ms (C++ member), 140 esp_http_client_config_t::tls_version (C++ member), 140 esp_http_client_config_t::transport_type (C++ member), 141 esp_http_client_config_t::url (C++ member), 139 esp_http_client_config_t::use_global_ca_store (C++ member), 141 esp_http_client_config_t::user_agent (C++ member), 140 esp_http_client_config_t::user_data (C++ member), 141 esp_http_client_config_t::username (C++ member), 139

esp_http_client_config_t::user_data (C++ member), 140 esp_http_client_config_t::user_agent (C++ member), 141 esp_http_client_config_t::use_global_ca_store (C++ member), 138 esp_http_client_config_t::url (C++ member), 138 esp_http_client_config_t::transport_type (C++ member), 138 esp_http_client_config_t::tls_version (C++ member), 138 esp_http_client_config_t::skip_cert_common_name_check (C++ member), 138 esp_http_client_config_t::query (C++ member), 138 esp_http_client_config_t::port (C++ member), 138 esp_http_client_config_t::path (C++ member), 138 esp_http_client_config_t::password (C++ member), 138 esp_http_client_config_t::method (C++ member), 138 esp_http_client_config_t::max_redirection_count (C++ member), 138 esp_http_client_config_t::max_authorization_retries (C++ member), 138 esp_http_client_config_t::keep_alive_interval (C++ member), 138 esp_http_client_config_t::keep_alive_idle (C++ member), 138 esp_http_client_config_t::keep_alive_enable (C++ member), 138 esp_http_client_config_t::keep_alive_count (C++ member), 138 esp_http_client_config_t::host (C++ member), 138 esp_http_client_config_t::if_name (C++ member), 138 esp_http_client_config_t::is_async (C++ member), 138 esp_http_client_event (C++ struct), 138 esp_http_client_event (C++ struct), 139 esp_http_client_event (C++ struct), 140

esp_http_client_delete_header (C++ function), 135 esp_http_client_get_status_code (C++ function), 134 esp_http_client_get_password (C++ function), 133 esp_http_client_get_post_field (C++ function), 133 esp_http_client_get_post_field (C++ function), 134 esp_http_client_get_header (C++ function), 134 esp_http_client_get_errno (C++ function), 134 esp_http_client_get_content_length (C++ function), 133 esp_http_client_get_chunk_length (C++ function), 133 esp_http_client_get_chunk_length (C++ function), 133

Index

(C++ function), 133

Submit Document Feedback
function), 136
esp_http_client_get_transport_type
(C++ function), 136
esp_http_client_get_url (C++ function), 138
esp_http_client_get_user_data (C++ function), 134
esp_http_client_get_username (C++ function), 133
esp_http_client_handle_t (C++ type), 142
esp_http_client_init (C++ function), 132
esp_http_client_is_chunked_response
(C++ function), 136
esp_http_client_is_complete_data_received
(C++ function), 137
esp_http_client_method_t (C++ enum), 144
esp_http_client_method_t::HTTP_METHOD_COPY
(C++ enumerator), 145
esp_http_client_method_t::HTTP_METHOD_DELETE
(C++ enumerator), 144
esp_http_client_method_t::HTTP_METHOD_GET
(C++ enumerator), 144
esp_http_client_method_t::HTTP_METHOD_HEAD
(C++ enumerator), 144
esp_http_client_method_t::HTTP_METHOD_LOCK
(C++ enumerator), 145
esp_http_client_method_t::HTTP_METHOD_MAX
(C++ enumerator), 145
esp_http_client_method_t::HTTP_METHOD.Move
(C++ enumerator), 145
esp_http_client_method_t::HTTP_METHOD_MOVE
(C++ enumerator), 145
esp_http_client_method_t::HTTP_METHOD_NOTIFY
(C++ enumerator), 144
esp_http_client_method_t::HTTP_METHOD_OPTIONS
(C++ function), 132
esp_http_client_method_t::HTTP_METHOD_OPTIONS
(C++ function), 132
esp_http_client_method_t::HTTP_METHOD_PATCH
(C++ function), 137
esp_http_client_method_t::HTTP_METHOD_POST
(C++ function), 135
esp_http_client_method_t::HTTP_METHOD_POST
(C++ function), 135
esp_http_client_method_t::HTTP_METHOD_PROPPATCH
(C++ function), 134
esp_http_client_method_t::HTTP_METHOD_PROPFIND
(C++ function), 133
esp_http_client_method_t::HTTP_METHOD_PUBLISH
(C++ function), 134
esp_http_client_method_t::HTTP_METHOD_PUBLISH
(C++ function), 134
esp_http_client_method_t::HTTP_METHOD_PROPPATCH
(C++ function), 134
esp_http_client_method_t::HTTP_METHOD_URI
(C++ function), 134
esp_http_client_method_t::HTTP_METHOD_URI
(C++ function), 134
esp_http_client_open (C++ function), 135
esp_http_client_open (C++ function), 135
esp_http_client_perform (C++ function), 132
esp_http_client_perform (C++ function), 132
esp_http_client.proto_ver_t (C++ enum), 144
esp_http_client.proto_ver_t::ESP_HTTP_CLIENT_TLS_1_0
(C++ enum), 144
esp_http_client.proto_ver_t::ESP_HTTP_CLIENT_TLS_1_1
(C++ enum), 144
esp_http_client.proto_ver_t::ESP_HTTP_CLIENT_TLS_1_2
(C++ enum), 144
esp_http_client_read (C++ function), 136
esp_http_client.redirect_response
(C++ function), 137
esp_http_server_event_data::fd
(C++ function), 192
esp_http_server_event_data::fd
(C++ function), 192
esp_http_server_event_data::header
(C++ function), 192
ESP_IP6_ADDR_BLOCK7 (C macro), 431
ESP_IP6_ADDR_BLOCK9 (C macro), 431
esp_ip6_addr_t (C++ type), 432
esp_ip6_addr_type_t (C++ enum), 432
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_GROUP
(C++ enumerator), 432
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_LINK_LOCAL
(C++ enumerator), 432
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_SITE_LOCAL
(C++ enumerator), 432
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_UNIQUE_LOCAL
(C++ enumerator), 432
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_V6LINK
(C++ enumerator), 432
esp_ip6_addr_type_t::ESP_IP6_ADDR_IS_V6LINKLOCAL
(C++ enumerator), 432
ESP_IP6ADDR_INIT (C macro), 432
est_ip_addr_t (C++ type), 432
ESP_IP_ADDR_ANY (C macro), 432
ESP_IPADDR_TYPE_ANY (C macro), 432
ESP_IPADDR_TYPE_V4 (C macro), 432
ESP_IPADDR_TYPE_V6 (C macro), 432
est_lcd_del_i80_bus (C++ function), 598
est_lcd_i2c_bus_handle_t (C++ struct), 603
nest_lcd_i80_bus_config_t (C++ struct), 601
nest_lcd_i80_bus_config_t::clk_src
(C++ member), 601
nest_lcd_i80_bus_config_t::data_gpio_num
(C++ member), 601
nest_lcd_i80_bus_config_t::dc_gpio_num
(C++ member), 601
nest_lcd_i80_bus_config_t::max_transfer_bytes
(C++ member), 601
nest_lcd_i80_bus_config_t::psram_trans_align
(C++ member), 601
nest_lcd_i80_bus_config_t::sram_trans_align
(C++ member), 601
nest_lcd_i80_bus_config_t::wr_gpio_num
(C++ member), 601
nest_lcd_i80_bus_handle_t (C++ type), 603
nest_lcd_new_i80_bus (C++ function), 598
nest_lcd_new_panel_i80_c (C macro), 602
nest_lcd_new_panel_i80_c::i2c (C++ function), 597
nest_lcd_new_panel_i80_c::i2c_v1 (C++ function), 597
nest_lcd_new_panel_i80_c::i2c_v2 (C++ function), 597
nest_lcd_new_panel_i80 (C++ function), 598
nest_lcd_new_panel_i80::spi (C++ function), 597
nest_lcd_panel_del (C++ function), 604
nest_lcd_panel_disp_off (C++ function), 605
nest_lcd_panel_disp_on_off (C++ function), 605
nest_lcd_panel_disp_sleep (C++ function), 605
nest_lcd_panel_draw_bitmap (C++ function), 604
nest_lcd_panel_handle_t (C++ type), 595
nest_lcd_panel_init (C++ function), 603
nest_lcd_panel_invert_color (C++ function), 605
nest_lcd_panel_i80_callbacks_t (C++ struct), 598
nest_lcd_panel_i80_callbacks_t::on_color_trans_done
(C++ member), 599
nest_lcd_panel_i80_callbacks_t::on_color_trans_done_cb_t
(C++ type), 603
nest_lcd_panel_i80_callbacks_t::UNIQUE_ID (C++ function), 596
nest_lcd_panel_i80_event_data_t
(C++ struct), 598
nest_lcd_panel_i80_handle_t (C++ type), 595
nest_lcd_panel_i80_i2c_config_t (C++ struct), 600
est_lcd_panel_i80_i2c_config_t::control_phase_byte
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::dc_bit_offset
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::dc_low_on_data
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::dev_addr
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::disable_control_phase
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::flags
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::lcd_cmd_bits
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::lcd_param_bits
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::on_color_trans_done
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::scl_speed_hz
(C++ member), 600
nest_lcd_panel_i80_i2c_config_t::user_ctx
(C++ member), 600
nest_lcd_panel_i80_config_t (C++ struct), 601
nest_lcd_panel_i80_config_t::cs_active_high
(C++ member), 601
nest_lcd_panel_i80_config_t::cs_high
(C++ member), 601
nest_lcd_panel_i80_config_t::dc_active_high
(C++ member), 601
nest_lcd_panel_i80_config_t::dc_low
(C++ member), 601
nest_lcd_panel_i80_config_t::df_active_high
(C++ member), 601
nest_lcd_panel_i80_config_t::reg_address
(C++ member), 601
nest_lcd_panel_i80_config_t::register
(C++ member), 601
Index

esp_lcd_panel_io_i80_config_t::lcd_cmd
esp_lcd_panel_io_tx_color (C++ function)

esp_lcd_panel_io_i80_config_t::lcd_reset
esp_lcd_panel_io_tx_param (C++ function)

esp_lcd_panel_io_i80_config_t::on_color
esp_lcd_panel_mirror (C++ function)

esp_lcd_panel_io_i80_config_t::pclk_ac
esp_lcd_panel_set_gap (C++ function)

esp_lcd_panel_io_i80_config_t::pclk_hz
esp_lcd_panel_swap_xy (C++ function)

esp_lcd_panel_io_i80_config_t::pclk_id
esp_light_sleep_start (C++ function)

esp_lcd_panel_io_i80_config_t::reverse
esp_local_ctrl_add_property (C++ function)

esp_lcd_panel_io_i80_config_t::swap_color_by
esp_local_ctrl_config::handlers (C++ member)

esp_lcd_panel_io_i80_config_t::trans_queue_depth
esp_local_ctrl_config::max_properties (C++ member)

esp_lcd_panel_io_i80_config_t::user_ctx
esp_local_ctrl_config::proto_sec (C++ member)

esp_lcd_panel_io_i80_config_t::user_evt
esp_local_ctrl_config::transport (C++ member)

esp_lcd_panel_io_i80_config_t::register_event_callbacks
esp_local_ctrl_config::transport_config (C++ function)

esp_lcd_panel_io_rx_param (C++ function)

esp_lcd_panel_io_spi_config_t (C++ struct)

esp_lcd_panel_io_spi_config_t::cs_gpio_num
esp_local_ctrl_add_property (C++ function)

esp_lcd_panel_io_spi_config_t::cs_high
esp_local_ctrl_get_transport_ble (C++ function)

esp_lcd_panel_io_spi_config_t::cs_gpio_num
esp_local_ctrl_get_transport_httpd (C++ function)

esp_lcd_panel_io_spi_config_t::dc_gpio_num
esp_local_ctrl_handlers (C++ struct)

esp_lcd_panel_io_spi_config_t::dc_low_on_data
esp_local_ctrl_handlers::get_prop_values

esp_lcd_panel_io_spi_config_t::flags
esp_local_ctrl_handlers::set_prop_values

esp_lcd_panel_io_spi_config_t::lcd_cmd_bits
esp_local_ctrl_handlers::usr_ctx (C++ member)

esp_lcd_panel_io_spi_config_t::lcd_param_bits
esp_local_ctrl_handlers::usr_ctx_free_fn

esp_lcd_panel_io_spi_config_t::octal_mode
esp_local_ctrl_prop::flags (C++ member)

esp_lcd_panel_io_spi_config_t::on_color_trans
esp_local_ctrl_prop::ctx (C++ member)

esp_lcd_panel_io_spi_config_t::pclk_hz
esp_local_ctrl_prop::name (C++ member)

esp_lcd_panel_io_spi_config_t::quad_mode
esp_local_ctrl_prop::size (C++ member)

esp_lcd_panel_io_spi_config_t::sio_mode
esp_local_ctrl_prop::type (C++ member)

esp_lcd_panel_io_spi_config_t::spi_mode
esp_local_ctrl_prop_t (C++ type)

esp_lcd_panel_io_spi_config_t::trans_queue_depth
esp_local_ctrl_prop_val (C++ struct)

esp_lcd_panel_io_spi_config_t::user_ctx
esp_local_ctrl_prop_val::data (C++ member)

Submit Document Feedback
Index

eesp_log_early_timestamp

\begin{verbatim}
member), 152
esp_local_ctrl_prop_val::size  (C++ type), 155
esp_local_ctrl_prop_val_t (C++ type), 155
esp_local_ctrl_proto_sec (C++ enum), 156
esp_local_ctrl_proto_sec::PROTOCOL_SEC (C++ enum), 156
esp_local_ctrl_proto_stop (C++ function), 154
esp_local_ctrl_proto_stop_t (C++ type), 155
esp_local_ctrl_proto_remove_property (C++ function), 150
esp_local_ctrl_proto_security1_params_t (C++ type), 155
esp_local_ctrl_proto_security2_params_t (C++ type), 155
esp_local_ctrl_set_handler (C++ function), 151
esp_local_ctrl_start (C++ function), 150
esp_local_ctrl_stop (C++ function), 150
ESP_LOCAL_CTRL_TRANSPORT_BLE (C macro), 155
esp_local_ctrl_transport_config_ble_t (C++ type), 155
esp_local_ctrl_transport_config_httpd_t (C++ type), 155
esp_local_ctrl_transport_config_t (C++ union), 151
esp_local_ctrl_transport_config_t::ble (C++ member), 151
esp_local_ctrl_transport_config_t::httpd (C++ member), 151
ESP_LOCAL_CTRL_TRANSPORT_HTTPD (C macro), 155
esp_local_ctrl_transport_t (C++ type), 155
ESP_LOG_BUFFER_CHAR (C macro), 1641
ESP_LOG_BUFFER_CHAR_LEVEL (C macro), 1640
ESP_LOG_BUFFER_HEX (C macro), 1641
ESP_LOG_BUFFER_HEX_LEVEL (C macro), 1640
ESP_LOGBUFFERHEXDUMP (C macro), 1641
ESP_LOG_EARLY_IMPL (C macro), 1642
esp_log_early_timestamp (C++ function), 1640
esp_log_get_level_master (C++ function), 1639
esp_log_get_level (C++ function), 1639
ESP_LOG_LEVEL (C macro), 1642
esp_log_level_get (C++ function), 1639
ESP_LOG_LEVEL_LOCAL (C macro), 1642
esp_log_level_set (C++ function), 1639
Esp_log_level_t (C++ enum), 1644
esp_log_level_t::ESP_LOG_DEBUG (C++ enumerator), 1644
esp_log_level_t::ESP_LOG_ERROR (C++ enumerator), 1644
esp_log_level_t::ESP_LOG_INFO (C++ enumerator), 1644
esp_log_level_t::ESP_LOG_NONE (C++ enumerator), 1644
esp_log_level_t::ESP_LOG_VERBOSE (C++ enumerator), 1644
esp_log_level_t::ESP_LOG_WARN (C++ enumerator), 1644
esp_log_set_level_master (C++ function), 1639
esp_log_system_timestamp (C++ function), 1640
esp_log_writev (C++ function), 1640
esp_log_write (C++ function), 1640
ESP_LOGD (C macro), 1642
ESP_LOGE (C macro), 1642
ESP_LOGI (C macro), 1642
ESP_LOGV (C macro), 1642
ESP_LOGW (C macro), 1642
esp_mac_addr_len_get (C++ function), 1652
esp_mac_type_t (C++ enum), 1652
esp_mac_type_t::ESP_MAC_BASE (C++ enumerator), 1653
esp_mac_type_t::ESP_MAC_BT (C++ enumerator), 1652
esp_mac_type_t::ESP_MAC_EFUSE_CUSTOM (C++ enumerator), 1653
esp_mac_type_t::ESP_MAC_EFUSE_EXT (C++ enumerator), 1653
esp_mac_type_t::ESP_MAC_EFUSE_FACTORY (C++ enumerator), 1653
esp_mac_type_t::ESP_MAC_ETH (C++ enumerator), 1652
esp_mac_type_t::ESP_MAC_IEEE802154 (C++ enumerator), 1652
esp_mac_type_t::ESP_MAC_WIFI_SOFTAP (C++ enumerator), 1652
esp_mac_type_t::ESP_MAC_WIFI_STA (C++ enumerator), 1652
ESP_MESH_ALLOW_ROOT_CONFLICTS (C++ function), 347
esp_mesh_available_txupQ_num (C++ function), 1644
\end{verbatim}
Index

esp_mesh_connect (C++ function), 243
esp_mesh_deinit (C++ function), 234
esp_mesh_delete_group_id (C++ function), 244

esp_mesh_disable_ps (C++ function), 249
esp_mesh_disconnect (C++ function), 247
esp_mesh_enable_ps (C++ function), 248
esp_mesh_fix_root (C++ function), 245
esp_mesh_flush_scan_result (C++ function), 247

esp_mesh_flush_upstream_packets (C++ function), 247

esp_mesh_get_active_duty_cycle (C++ function), 249

esp_mesh_get_ap_assoc_expire (C++ function), 242
esp_mesh_get_ap_authmode (C++ function), 240

esp_mesh_get_ap_connections (C++ function), 240
esp_mesh_get_capacity_num (C++ function), 245
esp_mesh_get_config (C++ function), 238
esp_mesh_get_group_list (C++ function), 244
esp_mesh_get_grid_num (C++ function), 244
esp_mesh_get_id (C++ function), 238
esp_mesh_get_ie_crypto_key (C++ function), 245
esp_mesh_get_layer (C++ function), 240
esp_mesh_get_max_layer (C++ function), 239
esp_mesh_get_network_duty_cycle (C++ function), 250

esp_mesh_get_non_mesh_connections (C++ function), 240
esp_mesh_get_parent_bssid (C++ function), 240
esp_mesh_get_router (C++ function), 238
esp_mesh_get_router_bssid (C++ function), 248

esp_mesh_get_routing_table (C++ function), 242
esp_mesh_get_routing_table_size (C++ function), 242
esp_mesh_get_running_active_duty_cycle (C++ function), 250
esp_mesh_get_rx_pending (C++ function), 243
esp_mesh_get_self_organized (C++ function), 241

esp_mesh_get_subnet_nodes_list (C++ function), 247
esp_mesh_get_subnet_nodes_num (C++ function), 247

esp_mesh_get_topology (C++ function), 248
esp_mesh_get_total_node_num (C++ function), 242
esp_mqtt_client_config_t::network_t::timeout_function

ESP_MQTT_CLIENT_RECONNECT

ESP_MQTT_CLIENT_PUBLISH

ESP_MQTT_CLIENT_INIT

ESP_MQTT_CLIENT_HANDLE

ESP_MQTT_CLIENT_ENQUEUE

ESP_MQTT_CLIENT_DISCONNECT

ESP_MQTT_CLIENT_DESTROY

ESP_MQTT_CLIENT_GET_OUTBOX_SIZE

ESP_MQTT_CLIENT_RECONNECT

ESP_MQTT_CLIENT_REGISTER_EVENT

Index
Espreport Systems

esp_mqtt_set_config(
    esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_V_5,
    esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_V_3_1_1,
    esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_V_3_1,
    esp_mqtt_protocol_ver_t::MQTT_PROTOCOL_UNDEFINED,

    esp_mqtt_protocol_ver_t
        （
            esp_mqtt_event_t::total_data_len,
            esp_mqtt_event_t::topic_len,
            esp_mqtt_event_t::topic
                （
                    esp_mqtt_event_t::retain,
                    （
                        esp_mqtt_event_t::qos,
                        （
                            esp_mqtt_event_t::msg_id,
                            esp_mqtt_event_t::event_id,
                            esp_mqtt_event_t::error_handle,
                            （
                                esp_mqtt_event_t::data_len,
                                esp_mqtt_event_t::current_data_offset,
                                esp_mqtt_event_t::client,
                                （
                                    esp_mqtt_event_t
                                        （
                                            esp_mqtt_event_id_t::MQTT_USER_EVENT,
                                            esp_mqtt_event_id_t::MQTT_EVENT_UNSUBSCRIBED,
                                            esp_mqtt_event_id_t::MQTT_EVENT_SUBSCRIBED,
                                            esp_mqtt_event_id_t::MQTT_EVENT_PUBLISHED,
                                            esp_mqtt_event_id_t::MQTT_EVENT_ERROR,
                                            esp_mqtt_event_id_t::MQTT_EVENT_DISCONNECTED,
                                            esp_mqtt_event_id_t::MQTT_EVENT_DELETED,
                                            esp_mqtt_event_id_t::MQTT_EVENT_DATA,
                                            esp_mqtt_event_id_t::MQTT_EVENT_CONNECTED,
                                            esp_mqtt_event_id_t::MQTT_EVENT_BEFORE_CONNECT,
                                            esp_mqtt_event_id_t::MQTT_EVENT_ANY,
                                ）
                            ）
                        ）
                    ）
                ）
            ）
        ）
    ）
)
esp_netif_deinit (C++ function), 407
esp_netif_destroy (C++ function), 407
esp_netif_destroy_default_wifi (C++ function), 435
esp_netif_dhcp_option_id_t (C++ enum), 428
esp_netif_dhcp_option_id_t::ESP_NETIF_DNS_Fallback (C++ enumerator), 427
esp_netif_dhcp_option_id_t::ESP_NETIF_DNS_MAIN (C++ enumerator), 427
esp_netif_dhcp_option_id_t::ESP_NETIF_DNS_MAX (C++ enumerator), 427
esp_netif_driver_base_t (C++ struct), 424
esp_netif_driver_base_s::ip_address leases_time_s::netif (C++ member), 424
esp_netif_driver_ifconfig (C++ struct), 424
esp_netif_driver_ifconfig::handle (C++ member), 425
esp_netif_driver_ifconfig::transmit (C++ member), 425
esp_netif_driver_ifconfig::transmit_wrap (C++ member), 425
esp_netif_driver_ifconfig_t (C++ type), 427
esp_netif_driver_ifconfig_t::ESP_NETIF_DHCP_CLIENT (C++ struct), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_DHCP_SERVER (C++ struct), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_FLAG_AUTOUP (C++ member), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_FLAG_EVENT_IP_MODIFIED (C++ member), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_FLAG_GARP (C++ member), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_FLAG_IS_BRIDGE (C++ member), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_FLAG_IS_PPP (C++ member), 429
esp_netif_driver_ifconfig_t::ESP_NETIF_FLAG_MLDv6_REPORT (C++ member), 429
esp_netif_drvflags_t (C++ type), 426
esp_netif_free_rx_buffer (C++ function), 439
esp_netif_get_all_ip6 (C++ function), 417
esp_netif_get_default_netif (C++ function), 410
esp_netif_get_desc (C++ function), 418
esp_netif_get_dns_info (C++ function), 416
esp_netif_get_event_id (C++ function), 418
esp_netif_get_flags (C++ function), 418
esp_netif_get_handle_from_ifkey (C++ function), 418
esp_netif_get_handle_from_netif_impl (C++ function), 438
esp_netif_get_hostname (C++ function), 411
esp_netif_get_ifkey (C++ function), 418
esp_netif_get_io_driver (C++ function), 418
esp_netif_get_ip6_global (C++ function), 417
esp_netif_get_ip6_linklocal (C++ function), 416
esp_netif_get_ip_info (C++ function), 411
esp_netif_get_ip_info (C++ function), 411
esp_netif_get_ip6_info_t (C++ function), 438
esp_netif_get_netif_impl (C++ function), 413
esp_netif_get_netif_impl_index (C++ function), 413
esp_netif_get_nr_of_ifs (C++ function), 419
esp_netif_get_old_ip_info (C++ function), 412
esp_netif_get_route_prio (C++ function), 418
esp_netif_htonl (C macro), 431
esp_netif_inherent_config (C++ struct), 424
esp_netif_inherent_config::bridge_info (C++ member), 424
esp_netif_inherent_config::flags (C++ member), 424
esp_netif_inherent_config::get_ip_event (C++ member), 424
esp_netif_inherent_config::if_desc (C++ member), 424
esp_netif_inherent_config::if_key (C++ member), 424
esp_netif_inherent_config::ip_info (C++ member), 424
esp_netif_inherent_config::lost_ip_event (C++ member), 424
esp_netif_inherent_config::mac (C++ member), 424
esp_netif_inherent_config::route_prio (C++ member), 424
esp_netif_inherent_config_t (C++ type), 426
ESP_NETIF_INHERENT_DEFAULT_OPENTHREAD (C macro), 398
esp_netif_init (C++ function), 407
esp_netif_ioctldriver_handle (C++ type), 427
esp_netif_ip4_makeu32 (C macro), 431
esp_netif_ip6_get_addr_type (C++ function), 430
esp_netif_ip6_info_t (C++ struct), 422
esp_netif_ip6_info_t::ip (C++ member), 422
esp_netif_ip6_info_t::netmask (C++ member), 422
esp_netif_is_netif_up (C++ function), 411
esp_netif_join_ip6_multicast_group (C++ function), 410
esp_netif_leave_ip6_multicast_group (C++ function), 410
esp_netif_napt_disable (C++ function), 413
esp_netif_napt_enable (C++ function), 413
esp_netif_netstack_buf_free (C++ function), 419
esp_netif_netstack_buf_ref (C++ function), 419
esp_netif_netstack_config_t (C++ type), 427
esp_netif_new (C++ function), 407
esp_netif_next (C++ function), 418
esp_netif_nextunsafe (C++ function), 419
esp_netif_pair_mac_ip_t (C++ struct), 425
esp_netif_pair_mac_ip_t::ip (C++ member), 425
esp_netif_pair_mac_ip_t::mac (C++ member), 425
esp_netif_receive (C++ function), 408
esp_netif_receive_ctx (C++ type), 427
esp_netif_set_default_netif (C++ function), 410
esp_netif_set_dns_info (C++ function), 415
esp_netif_set_driver_config (C++ function), 407
esp_netif_set_hostname (C++ function), 411
esp_netif_set_ip4_addr (C++ function), 417
esp_netif_set_ip_info (C++ function), 412
esp_netif_set_link_speed (C++ function), 438
esp_netif_set_mac (C++ function), 411
esp_netif_set_old_ip_info (C++ function), 412
ESP_NETIF_SNTP_DEFAULT_CONFIG (C macro), 421
ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE (C macro), 421
esp_netif_str_to_ip (C++ function), 417
esp_netif_str_to_ip6 (C++ function), 417
esp_netif_t (C++ type), 426
ESP_NETIF_UDP_API (C++ function), 419
esp_netif_transmit (C++ function), 438
esp_netif_transmit_wrap (C++ function), 438
esp_nox_add_peer (C++ function), 223
esp_nox_deinit (C++ function), 222
esp_now_del_peer (C++ function), 223
ESP_NOW_ETH_ALEN (C macro), 227
esp_now_fetch_peer (C++ function), 224
esp_now_get_peer (C++ function), 224
esp_now_get_peer_num (C++ function), 224
esp_now_get_version (C++ function), 222
esp_now_init (C++ function), 222
esp_now_is_peer_exist (C++ function), 224
ESP_NOW_KEY_LEN (C macro), 227
ESP_NOW_MAX_DATA_LEN (C macro), 227
ESP_NOW_MAX_TOTAL_PEER_NUM (C macro), 227
esp_now_mod_peer (C++ function), 223
esp_now_peer_info (C++ struct), 225
esp_now_peer_info::channel (C++ member), 225
esp_now_peer_info::decrypt (C++ member), 225
esp_now_peer_info::ifidx (C++ member), 225
esp_now_peer_info::lmk (C++ member), 225
esp_now_peer_info::peer_addr (C++ member), 225
esp_now_peer_info::priv (C++ member), 226
esp_now_peer_info_t (C++ type), 227
esp_now_peer_num (C++ struct), 226
esp_now_peer_num::encrypt_num (C++ member), 226
esp_now_peer_num::total_num (C++ member), 226
esp_now_peer_num_t (C++ type), 227
esp_now_rate_config (C++ struct), 226
esp_now_rate_config::dcm (C++ member), 226
esp_now_rate_config::encryp (C++ member), 226
esp_now_rate_config::encrypt (C++ member), 226
esp_now_rate_config::phymode (C++ member), 226
esp_now_rate_config::rate (C++ member), 226
esp_now_rate_config_t (C++ type), 228
esp_now_recv_cb_t (C++ type), 228
esp_now_recv_info (C++ struct), 226
esp_now_recv_info::des_addr (C++ member), 226
esp_now_recv_info::ifidx (C++ member), 226
esp_now_recv_info::src_addr (C++ member), 226
esp_now_register_recv_cb (C++ function), 222
esp_now_register_send_cb (C++ function), 222
esp_now_send (C++ function), 222
esp_now_send.cb_t (C++ type), 228
esp_now_send_status_t (C++ enum), 228
esp_now_send_status_t::ESP_NOW_SEND_FAIL (C++ enumerator), 228
esp_now_send_status_t::ESP_NOW_SEND_SUCCESS (C++ enumerator), 228
esp_now_set_peer_rate_config (C++ function), 224
esp_now_set_pmk (C++ function), 225
esp_now_set_wake_window (C++ function), 225
esp_now_unregister_recv_cb (C++ function), 222
esp_now_unregister_recvcb (C++ function), 222
esp_now_unregister_send_cb (C++ function), 222
ESP_OK (C macro), 1415
ESP_OK_EFUSE_CNT (C macro), 1412
esp_openthread_auto_start (C++ function), 391
esp_openthread_border_router_deinit (C++ function), 399
esp_openthread_border_router_init (C++ function), 399
esp_openthread_deinit (C++ function), 391
esp_openthread_event_t (C++ enum), 395
esp_openthread_event_t::OPENTHREAD_EVENT_ATTACHMENT (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_DETACHMENT (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_GOT_IP6 (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_IF_DOWN (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_IF_UP (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_LOST_IP6 (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_MULTICAST (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_MULTICAST (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_ROLE_CHANGE (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_SET_DNS (C++ enumerator), 396
esp_openthread_event_t::OPENTHREAD_EVENT_START (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_STOP (C++ enumerator), 395
esp_openthread_event_t::OPENTHREAD_EVENT_TREL_ADD (C++ enumerator), 396
esp_openthread_event_t::OPENTHREAD_EVENT_TREL_REMOVED (C++ enumerator), 396
esp_openthread_get_backbone_netif (C++ function), 399
esp_openthread_get_instance (C++ function), 391
esp_openthread_get_netif (C++ function), 398

Index

- esp_openthread_host_connection_config_t
  - (C++ struct), 394
- esp_openthread_host_connection_config_t::host::netif_mode
  - (C++ member), 394
- esp_openthread_host_connection_config_t::host::storage_partition_name
  - (C++ member), 394
- esp_openthread_host_connection_config_t::max_fd
  - (C++ member), 394
- esp_openthread_host_connection_mode_t::HOST_CONNECTION_MODE_RCP_UART
  - (C++ enum), 396
- esp_openthread_host_connection_mode_t::HOST_CONNECTION_MODE_SPI_RCP
  - (C++ enum), 396
- esp_openthread_init
  - (C++ function), 390
- esp_openthread_launch_mainloop
  - (C++ function), 391
- esp_openthread_lock_acquire
  - (C++ function), 397
- esp_openthread_lock_deinit
  - (C++ function), 397
- esp_openthread_lock_init
  - (C++ function), 397
- esp_openthread_lock_release
  - (C++ function), 397
- esp_openthread_mainloop_context_t
  - (C++ struct), 392
- esp_openthread_mainloop_context_t::error
  - (C++ member), 392
- esp_openthread_mainloop_context_t::max
  - (C++ member), 392
- esp_openthread_mainloop_context_t::read
  - (C++ member), 392
- esp_openthread_mainloop_context_t::timemap
  - (C++ member), 393
- esp_openthread_netif_glue_deinit
  - (C++ function), 398
- esp_openthread_netif_glue_init
  - (C++ function), 398
- esp_openthread_platform_config_t
  - (C++ struct), 394
- esp_openthread_platform_config_t::host
  - (C++ member), 395
- esp_openthread_platform_config_t::port
  - (C++ member), 395
- esp_openthread_platform_config_t::radio
  - (C++ member), 395


Index

esp_phy_cbw40m_en (C++ function), 1993  esp_phy_wifi_rate_t::PHY_RATE_2M (C++ enumerator), 1995
esp_phy_common_clock_disable (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_2M (C++ enumerator), 1995
esp_phy_common_clock_enable (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_36M (C++ enumerator), 1995
esp_phy_disable (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_48M (C++ enumerator), 1995
esp_phy_enable (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_54M (C++ enumerator), 1995
esp_phy_erase_cal_data_in_nvs (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_5M5 (C++ enumerator), 1995
esp_phy_get_init_data (C++ function), 1989  esp_phy_wifi_rate_t::PHY_RATE_6M (C++ enumerator), 1995
esp_phy_get_rx_result (C++ function), 1994  esp_phy_wifi_rate_t::PHY_RATE_7M5 (C++ enumerator), 1995
esp_phy_init_data_t::params (C++ member), 1991  esp_phy_wifi_rate_t::PHY_RATE_8M (C++ enumerator), 1995
esp_phy_load_cal_and_init (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_MCS0 (C++ enumerator), 1995
esp_phy_load_cal_data_from_nvs (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_MCS1 (C++ enumerator), 1995
esp_phy_modem_deinit (C++ function), 1990  esp_phy_wifi_rate_t::PHY_RATE_MCS2 (C++ enumerator), 1995
esp_phy_modem_t (C++ enum), 1992  esp_phy_wifi_rate_t::PHY_RATE_MCS3 (C++ enumerator), 1995
esp_phy_modem_t::PHY_MODEM_BT (C++ enum), 1992  esp_phy_wifi_rate_t::PHY_RATE_MCS4 (C++ enumerator), 1995
esp_phy_modem_t::PHY_MODEM_IEEE802154 (C++ enum), 1992  esp_phy_wifi_rate_t::PHY_RATE_MCS5 (C++ enumerator), 1995
esp_phy_modem_t::PHY_MODEM_WIFI (C++ enum), 1992  esp_phy_wifi_rate_t::PHY_RATE_MCS6 (C++ enumerator), 1995
esp_phy_release_init_data (C++ function), 1989  esp_phy_wifi_rate_t::PHY_RATE_MCS7 (C++ enumerator), 1995
esp_phy_rf_get_on_ts (C++ function), 1990  esp_phy_wifi_rate_t::PHY_WIFI_RATE_MAX (C++ enumerator), 1995
esp_phy_rftest_config (C++ function), 1992  esp_phy_wifi_tx (C++ function), 1993
esp_phy_rftest_init (C++ function), 1993  esp_phy_wifi_tx_t::cb_args (C++ member), 208
esp_phy_rx_result_t::phy_rx_correct_count (C++ member), 1994  esp_phy_wifi_callbacks_t::on_ping_end (C++ member), 209
esp_phy_rx_result_t::phy_rx_result_flag (C++ member), 1994  esp_phy_wifi_callbacks_t::on_ping_end (C++ member), 209
esp_phy_rx_result_t::phy_rx_rssi (C++ member), 1994  esp_phy_wifi_callbacks_t::on_ping_success (C++ member), 209
esp_phy_rx_result_t::phy_rx_total_count (C++ member), 1994  esp_phy_wifi_callbacks_t::on_ping_timeout (C++ member), 209
esp_phy_test_cal_data_to_nvs (C++ function), 1990  esp_phy_config_t (C++ struct), 209
esp_phy_test_start_stop (C++ function), 1993  esp_phy_config_t::count (C++ member), 209
esp_phy_tx_contin (C++ function), 1991  esp_phy_config_t::data_size (C++ member), 209
esp_phy_update_country_info (C++ function), 1991  esp_phy_config_t::interface (C++ member), 209
esp_phy_wifi_rate_t (C++ enum), 1995  esp_phy_config_t::interval_ms (C++ member), 209
esp_phy_wifi_rate_t::PHY_RATE_11M (C++ enumerator), 1995  esp_phy_config_t::target_addr (C++ member), 209
esp_phy_wifi_rate_t::PHY_RATE_12M (C++ enumerator), 1995  esp_phy_config_t::task_prio (C++ member), 209
esp_phy_wifi_rate_t::PHY_RATE_18M (C++ enumerator), 1995
esp_phy_wifi_rate_t::PHY_RATE_1M (C++ enumerator), 1995
esp_phy_wifi_rate_t::PHY_RATE_24M (C++ enumerator), 1995

Espressif Systems 2251 Release master

Submit Document Feedback
Index

esp_ping_config_t::task_stack_size (C++ member), 209
esp_ping_config_t::timeout_ms (C++ member), 209
esp_ping_config_t::tos (C++ member), 209
ESP_PM_COUNT_INFINITE (Cmacro), 210
ESP_PM_DEFAULT_CONFIG (Cmacro), 210
esp_ping_delete_session (C++ function), 208
esp_ping_get_profile (C++ function), 208
esp_ping_handle_t (C++ type), 210
esp_ping_new_session (C++ function), 208
esp_ping_profile_t (C++ enum), 210
esp_ping_profile_t::ESP_PIN_PROFILE_DURABLE (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_IPADD (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_REPLY (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_REQUEST (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_SEQNUM (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_SIZE (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_TIMEGAP (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_TOS (C++ enumerator), 210
esp_ping_profile_t::ESP_PIN_PROFILE_TTL (C++ enumerator), 210
esp_ping_start (C++ function), 208
esp_ping_stop (C++ function), 208
esp_pm_config_esp32s3_t (C++ type), 1682
esp_pm_config_esp32c6_t (C++ type), 1682
esp_pm_config_esp32c2_t (C++ type), 1682
esp_pm_config_esp32c3_t (C++ type), 1683
esp_pm_config_esp32s2_t (C++ type), 1682
esp_pm_config_esp32s3_t (C++ type), 1682
esp_pm_reset (C++ struct), 1682
esp_pm_config_t::light_sleep_enable (C++ member), 1682
esp_pm_config_t::max_freq_mhz (C++ member), 1682
esp_pm_config_t::min_freq_mhz (C++ member), 1682
esp_pm_configure (C++ function), 1680
esp_pm_dump_locks (C++ function), 1682
esp_pm_get_configuration (C++ function), 1680
esp_pm_lock_acquire (C++ function), 1681
esp_pm_lock_create (C++ function), 1681
esp_pm_lock_delete (C++ function), 1681
esp_pm_lock_handler (C++ type), 1683
esp_pm_lock_release (C++ function), 1681
esp_pm_lock_type_t (C++ enum), 1683
esp_pm_lock_type_t::ESP_PM_APB_FREQ_MAX (C++ enumerator), 1683
esp_pm_lock_type_t::ESP_PM_CPU_FREQ_MAX (C++ enumerator), 1683
esp_pm_lock_type_t::ESP_PM_NO_LIGHT_SLEEP (C++ enumerator), 1683
esp_pm_lock_type_t::ESP_PM_SET_FREQ (C++ function), 1683
esp_pm_lock_release (C++ function), 1681
esp_pm_lock_delete (C++ function), 1681
esp_pm_lock_handler (C++ type), 1683
esp_pm_lock_release (C++ function), 1681
esp_pm_lock_type_t (C++ enum), 1683
esp_pm_lock_type_t::ESP_PM_APB_FREQ_MAX (C++ enumerator), 1683
esp_pm_lock_type_t::ESP_PM_CPU_FREQ_MAX (C++ enumerator), 1683
esp_pm_lock_type_t::ESP_PM_NO_LIGHT_SLEEP (C++ enumerator), 1683
esp_pm_lock_type_t::ESP_PM_SET_FREQ (C++ function), 1683
esp_pm_lock_release (C++ function), 1681
esp_pm_lock_delete (C++ function), 1681
esp_pm_lock_handler (C++ type), 1683
esp_pm_lock_release (C++ function), 1681
esp_pm_lock_type_t (C++ enum), 1683
esp_reset_reason_t::ESP_RST_NMI (C++
function), 1649
esp_restart (C++ function), 1648
ESP_RETURN_ON_ERROR (C macro), 1414
ESP_RETURN_ON_ERROR_ISR (C macro), 1414
ESP_RETURN_ON_FALSE (C macro), 1414
esp_rom_delay_us (C++ function), 1628
esp_rom_get_cpu_ticks_per_us (C++ func-
tion), 1628
esp_rom_get_reset_reason (C++ function),
1628
esp_rom_install_channel_putchar (C++ func-
tion), 1628
esp_rom_install_uart_putchar (C++ function),
1628
esp_rom_printf (C++ function), 1628
esp_rom_set_cpu_ticks_per_us (C++ function),
1628
esp_rom_software_reset_cpu (C++ function),
1627
esp_rom_software_reset_system (C++ function),
1627
esp_rrm_is_rrm_supported_connection (C++
function), 1412
esp_rrm_send_neighbor_rep_request (C++
function), 1412
esp_secure_boot_key_digests_t::key_digests
(C++ function), 1412
esp_secure_boot_key_digests_t::key_digests
(C++ member), 1412
esp_secure_boot_key_digests_t::key_digests
(C++ structure), 1412
esp_secure_boot_key_digests_t::key_digests
(C++ structure), 1412
esp_secure_boot_key_digests_t::key_digests
(C++ member), 1412
esp_secure_boot_key_digests_t::key_digests
(C++ function), 1412
esp_secure_boot_read_key_digests (C++ func-
tion), 1411
esp_set_deep_sleep_mode (C++ function),
1702
esp_set_deep_sleep_mode_default_entry
(C++ function), 1702
esp_sleep_enable_timer_wakeup (C++ func-
tion), 1699
esp_sleep_enable_timer_wakeup (C++ func-
tion), 1699
esp_sleep_enable_touchpad_wakeup (C++
function), 1697
esp_sleep_enable_touchpad_wakeup (C++
function), 1697
esp_sleep_enable_uart_wakeup (C++ function),
1699
esp_sleep_enable_ulp_wakeup (C++ function),
1697
esp_sleep_enable_wifi_beacon_wakeup
(C++ function), 1700
esp_sleep_enable_wifi_wake (C++ function),
1700
esp_sleep_enable_wifi_wake (C++ function),
1700
esp_sleep_ext1_wakeup_mode_t (C++ enum),
1703
esp_sleep_ext1_wakeup_mode_t::ESP_EXT1_WAKEUP_ALL
(C++ enumerator), 1703
esp_sleep_ext1_wakeup_mode_t::ESP_EXT1_WAKEUP_ALL
(C++ enumerator), 1703
esp_sleep_ext1_wakeup_mode_t::ESP_EXT1_WAKEUP_ANY
(C++ enumerator), 1703
esp_sleep_ext1_wakeup_mode_t::ESP_EXT1_WAKEUP_ANY
(C++ enumerator), 1703
esp_sleep_get_ext1_wakeup_status (C++
function), 1700
esp_sleep_get_ext1_wakeup_status (C++
function), 1700
esp_sleep_get_ext1_wakeup_status (C++
function), 1697
esp_sleep_get_wakeup_cause (C++ function),
1702
esp_sleep_is_valid_wakeup_gpio (C++
function), 1698
esp_sleep_mode_t (C++ enum), 1705
esp_sleep_mode_t::ESP_SLEEP_MODE_DEEP_SLEEP
(C++ enumerator), 1705
esp_sleep_mode_t::ESP_SLEEP_MODE_DEEP_SLEEP
(C++ enumerator), 1705
esp_sleep_mode_t::ESP_SLEEP_MODE_LIGHT_SLEEP
(C++ enumerator), 1705
esp_sleep_mode_t::ESP_SLEEP_MODE_LIGHT_SLEEP
(C++ enumerator), 1705
esp_sleep_pd_domain (C++ function), 1700
esp_sleep_pd_domain_t (C++ enum), 1703
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_MAX
(C++ function), 1702
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_MAX
(C++ function), 1702
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_FAST
(C++ function), 1702
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_FAST
(C++ function), 1702
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_PERIPH
(C++ function), 1702
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_RTC_PERIPH
(C++ function), 1702
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_VDDSDIO
(C++ function), 1704
esp_sleep_pd_domain_t::ESP_PD_DOMAIN_VDDSDIO
(C++ function), 1704
esp_sleep_pd_option_t (C++ enum), 1704
esp_sleep_pd_option_t::ESP_PD_OPTION_AUTO
(C++ function), 1704
esp_sleep_pd_option_t::ESP_PD_OPTION_AUTO
(C++ function), 1704
esp_sleep_pd_option_t::ESP_PD_OPTION_OFF
(C++ function), 1704
esp_sleep_pd_option_t::ESP_PD_OPTION_OFF
(C++ function), 1704
esp_sleep_source_t (C++ enum), 1704

Espressif Systems 2253

Submit Document Feedback
esp_sleep_source_t::ESP_SLEEP_WAKEUP_A
esp_sntp_config::wait_for_sync
(\texttt{C++ function}), 1724
esp_sntp_config::wait_for_sync
(\texttt{C++ enumerator}), 1704
esp_sntp_config::wait_for_sync
(\texttt{C++ macro}), 1725
esp_sntp_config::wait_for_sync
(\texttt{C++ function}), 1724
esp_sntp_config::sync_cb
(\texttt{C++ function}), 1724
esp_sntp_config::start
(\texttt{C++ function}), 1724
esp_sntp_config::smooth_sync
(\texttt{C++ function}), 1724
esp_sntp_config::servers
(\texttt{C++ function}), 1724
esp_sntp_config::servers
(\texttt{C++ member}), 1705
esp_sntp_config::num_of_servers
(\texttt{C++ function}), 1724
esp_sntp_config::num_of_servers
(\texttt{C++ member}), 1705
esp_sntp_config::renew_servers_after_notification
(\texttt{C++ function}), 1724
esp_sntp_config::renew_servers_after_notification
(\texttt{C++ member}), 1705
esp_sntp_config::server_from_dhcp
(\texttt{C++ function}), 1724
esp_sntp_config::server_from_dhcp
(\texttt{C++ member}), 1705
esp_sntp_config::servers
(\texttt{C++ function}), 1724
esp_sntp_config::servers
(\texttt{C++ member}), 1705
esp_sntp_config::smooth_sync
(\texttt{C++ function}), 1724
esp_sntp_config::smooth_sync
(\texttt{C++ member}), 1705
esp_sntp_config::start
(\texttt{C++ function}), 1724
esp_sntp_config::start
(\texttt{C++ member}), 1705
esp_sntp_config::sync_cb
(\texttt{C++ function}), 1724
esp_sntp_config::sync_cb
(\texttt{C++ member}), 1705
esp_sntp_config::wait_for_sync
(\texttt{C++ function}), 1724
esp_sntp_config::wait_for_sync
(\texttt{C++ member}), 1705

Index
Index

esp_sysview_vprintf (C++ function), 1368
esp_task_wdt_add (C++ function), 1785
esp_task_wdt_add_user (C++ function), 1786
esp_task_wdt_config_t::idle_core_mask (C++ member), 1787
esp_task_wdt_config_t::timeout_ms (C++ member), 1787
esp_task_wdt_config_t::trigger_panic (C++ member), 1787
esp_task_wdt_deinit (C++ function), 1785
esp_task_wdt_delete (C++ function), 1786
esp_task_wdt_delete_user (C++ function), 1786
esp_task_wdt_init (C++ function), 1785
esp_task_isr_user_handler (C++ function), 1785
esp_task_wdt_reconfigure (C++ function), 1787
esp_task_wdt_reset (C++ function), 1786
esp_task_wdt_reset_user (C++ function), 1786
esp_task_wdt_status (C++ function), 1787
esp_task_wdt_user_handle_t (C++ type), 1787
esp_timer_cb_t (C++ type), 1627
esp_timer_create (C++ function), 1623
esp_timer_create_args_t (C++ struct), 1626
esp_timer_create_args_t::arg (C++ member), 1626
esp_timer_create_args_t::callback (C++ member), 1626
esp_timer_create_args_t::dispatch_method (C++ member), 1626
esp_timer_create_args_t::name (C++ member), 1626
esp_timer_create_args_t::skip_unhandled_event (C++ member), 1626
esp_timer_deinit (C++ function), 1623
esp_timer_delete (C++ function), 1624
esp_timer_dispatch_t (C++ enum), 1627
esp_timer_dispatch_t::ESP_TIMER_MAX (C++ enumerator), 1627
esp_timer_dispatch_t::ESP_TIMER_TASK (C++ enumerator), 1627
esp_timer_dump (C++ function), 1625
esp_timer_early_init (C++ function), 1622
esp_timer_get_expiry_time (C++ function), 1625
esp_timer_get_next_alarm (C++ function), 1624
esp_timer_get_next_alarm_for_wake_up (C++ function), 1624
esp_timer_get_period (C++ function), 1625
esp_timer_get_time (C++ function), 1624
esp_timer_handle_t (C++ type), 1627
esp_timer_init (C++ function), 1622
esp_timer_is_active (C++ function), 1626
esp_timer_isr_dispatch_need_yield (C++ function), 1625
esp_timer_new_etm_alarm_event (C++ function), 1626
esp_timer_restart (C++ function), 1624
esp_timer_start_once (C++ function), 1623
esp_timer_start_periodic (C++ function), 1624
esp_timer_stop (C++ function), 1624
esp_tls_addr_family (C++ enum), 125
esp_tls_addr_family::ESP_TLS_AF_INET (C++ enumerator), 125
esp_tls_addr_family::ESP_TLS_AF_INET6 (C++ enumerator), 125
esp_tls_addr_family::ESP_TLS_AF_UNSPEC (C++ enumerator), 125
esp_tls_addr_family_t (C++ type), 124
esp_tls_cfg (C++ struct), 119
esp_tls_cfg::addr_family (C++ member), 122
esp_tls_cfg::alpn_protos (C++ member), 120
esp_tls_cfg::caceret_buf (C++ member), 120
esp_tls_cfg::caceret_bytes (C++ member), 120
esp_tls_cfg::caceret_pem_buf (C++ member), 120
esp_tls_cfg::caceret_pem_bytes (C++ member), 120
esp_tls_cfg::ciphersuites_list (C++ member), 122
esp_tls_cfg::clientcert_buf (C++ member), 120
esp_tls_cfg::clientcert_bytes (C++ member), 120
esp_tls_cfg::clientcert_pem_buf (C++ member), 120
esp_tls_cfg::clientcert_pem_bytes (C++ member), 120
esp_tls_cfg::clientkey_buf (C++ member), 120
esp_tls_cfg::clientkey_bytes (C++ member), 121
esp_tls_cfg::clientkey_password (C++ member), 121
esp_tls_cfg::clientkey_password_len (C++ member), 121
esp_tls_cfg::clientkey_pem_buf (C++ member), 121
esp_tls_cfg::clientkey_pem_bytes (C++ member), 121
esp_tls_cfg::common_name (C++ member), 121
esp_tls_cfg::crt_bundle_attach (C++ member), 122
esp_tls_cfg::ds_data (C++ member), 122
esp_tls_cfg::ecdsa_key_efuse_blk (C++ member), 121
esp_tls_cfg::if_name (C++ member), 122
esp_tls_cfg::is_plain_tcp (C++ member), 122
esp_tls_cfg::keep_alive_cfg (C++ member), 121
esp_tls_cfg::non_block (C++ member), 121
esp_tls_cfg::psk_hint_key (C++ member), 121
esp_tls_cfg::skip_common_name (C++ member), 121
esp_tls_cfg::timeout_ms (C++ member), 122
esp_tls_cfg::tls_version (C++ member), 122
esp_tls_cfg::use_ecdsa_peripheral (C++ member), 121
esp_tls_cfg::use_global_ca_store (C++ member), 121
esp_tls_cfg::use_secure_element (C++ member), 121
esp_tls_cfg_server (C++ struct), 122
esp_tls_cfg_server::alpn_protos member), 122
esp_tls_cfg_server::cacert_buf member), 122
esp_tls_cfg_server::cacert_bytes member), 122
esp_tls_cfg_server::cacert_pem_buf (C++ member), 122
esp_tls_cfg_server::cacert_pem_bytes (C++ member), 122
esp_tls_cfg_server::ecdsa_key_efuse_blk (C++ member), 123
esp_tls_cfg_server::servercert_buf (C++ member), 123
esp_tls_cfg_server::servercert_bytes (C++ member), 123
esp_tls_cfg_server::servercert_pem_buf (C++ member), 123
esp_tls_cfg_server::servercert_pem_bytes (C++ member), 123
esp_tls_cfg_server::serverkey_buf (C++ member), 123
esp_tls_cfg_server::serverkey_bytes (C++ member), 123
esp_tls_cfg_server::serverkey_password (C++ member), 123
esp_tls_cfg_server::serverkey_password_len (C++ member), 123
esp_tls_cfg_server::serverkey_pem_buf (C++ member), 123
esp_tls_cfg_server::serverkey_pem_bytes (C++ member), 123
esp_tls_cfg_server::use_ecdsa_peripheral (C++ member), 123
esp_tls_cfg_server::use_secure_element (C++ member), 123
esp_tls_cfg_server::userdata (C++ member), 123
esp_tls_server_session_tickets_free (C++ function), 114
esp_tls_server_session_tickets_init (C++ function), 114
esp_tls_server_session_tickets_init (C++ function), 114
esp_tls_server_t (C++ type), 124
esp_tls_t (C++ type), 124
esp_tls_conn_destroy (C++ function), 116
esp_tls_conn_http_new (C++ function), 114
esp_tls_conn_http_new_async (C++ function), 115
esp_tls_conn_http_new_async (C++ function), 115
esp_tls_conn_new (C++ function), 114
esp_tls_conn_new (C++ function), 115
esp_tls_conn_state (C++ enum), 124
esp_tls_conn_state::ESP_TLS_CONNECTION (C++ enumerator), 124
esp_tls_conn_state::ESP_TLS_DONE (C++ enumerator), 124
esp_tls_conn_state::ESP_TLS_FAIL (C++ enumerator), 124
esp_tls_conn_state::ESP_TLS_HANDSHAKE (C++ enumerator), 124
esp_tls_conn_state::ESP_TLS_INIT (C++ enumerator), 124
esp_tls_conn_state_t (C++ type), 123
esp_tls_conn_write (C++ function), 115
ESP_TLS_ERR_SSL_TIMEOUT (C macro), 128
ESP_TLS_ERR_SSL_WANT_WRITE (C macro), 128
ESP_TLS_ERR_SSL_WANT_WRITE (C macro), 128
esp_tls_error_handle_t (C++ type), 128
esp_tls_error_type_t (C++ enum), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_ESP (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MAX (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MBEDTLS (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MBEDTLS_CIPHER_SUITE (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MBEDTLS_CIPHER_SUITE (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_MBEDTLS_CIPHER_SUITE (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_UNKNOWN (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_WOLFSSL (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_WOLFSSL_CIPHER_SUITE (C++ enumerator), 128
esp_tls_error_type_t::ESP_TLS_ERR_TYPE_WOLFSSL_CIPHER_SUITE (C++ enumerator), 128
esp_tls_free_global_ca_store (C++ function), 117
esp_tls_get_and_clear_error_type (C++ function), 117
esp_tls_get_and_clear_last_error (C++ function), 117
esp_tls_get_bytes_avail (C++ function), 116
esp_tls_get_ciphersuites_list (C++ function), 118
Index

esp_tls_get_conn_sockfd (C++ function), 116
esp_tls_get_conn_state (C++ function), 116
esp_tls_get_error_handle (C++ function), 118
esp_tls_get_global_ca_store (C++ function), 118
esp_tls_get_ssl_context (C++ function), 117
esp_tls_init (C++ function), 114
esp_tls_init_global_ca_store (C++ function), 117
esp_tls_last_error (C++ struct), 125
esp_tls_last_error::esp_tls_error_code
(C++ member), 126
esp_tls_last_error::esp_tls_flags
(C++ member), 126
esp_tls_last_error::last_error
(C++ member), 125
esp_tls_plain_tcp_connect (C++ type), 128
esp_tls_proto_ver_t (C++ enum), 125
esp_tls_proto_ver_t::ESP_TLS_VER_ANY
(C++ enumerator), 125
esp_tls_proto_ver_t::ESP_TLS_VER_TLS_1_2
(C++ enumerator), 125
esp_tls_proto_ver_t::ESP_TLS_VER_TLS_1_3
(C++ enumerator), 125
esp_tls_role (C++ enum), 124
esp_tls_role::ESP_TLS_CLIENT
(C++ enumerator), 124
esp_tls_role::ESP_TLS_SERVER
(C++ enumerator), 125
esp_tls_role_t (C++ type), 123
esp_tls_server_session_create (C++ function), 118
esp_tls_server_session_delete (C++ function), 118
esp_tls_set_conn_sockfd (C++ function), 116
esp_tls_set_conn_state (C++ function), 116
esp_tls_set_global_ca_store (C++ function), 117
esp_tls_t (C++ type), 124
esp_unregister_shutdown_handler (C++ function), 1647
esp_vendor_ie_cb_t (C++ type), 293
esp_vfs_close (C++ function), 1340
esp_vfs_dev_uart_port_set_rx_line_endings
(C++ function), 1350
esp_vfs_dev_uart_port_set_tx_line_endings
(C++ function), 1350
esp_vfs_dev_uart_register (C++ function), 1349
esp_vfs_dev_uart_set_rx_line_endings
(C++ function), 1349
esp_vfs_dev_uart_set_tx_line_endings
(C++ function), 1349
esp_vfs_dev_uart_use_driver (C++ function), 1350
esp_vfs_dev_uart_use_nonblocking (C++ function), 1350
ESP_VFS_EVENTD_CONFIG_DEFAULT (C macro), 1352
esp_vfs_eventfd_config_t (C++ struct), 1351
esp_vfs_eventfd_config_t::max_fds
(C++ member), 1352
esp_vfs_eventfd_register (C++ function), 1351
esp_vfs_eventfd_unregister (C++ function), 1351
esp_vfs_fat_info (C++ function), 1274
esp_vfs_fat_mount_config_t (C++ struct), 1274
esp_vfs_fat_mount_config_t::allocation_unit_size
(C++ member), 1274
esp_vfs_fat_mount_config_t::disk_status_check_enable
(C++ member), 1274
esp_vfs_fat_mount_config_t::format_if_mount_failed
(C++ member), 1274
esp_vfs_fat_mount_config_t::max_files
(C++ member), 1274
esp_vfs_fat_register (C++ function), 1269
esp_vfs_fat_sdcard_format (C++ function), 1272
esp_vfs_fat_sdcard_unmount (C++ function), 1272
esp_vfs_fat_sdmmc_mount (C++ function), 1270
esp_vfs_fat_sdmmc_mount_config_t (C++ type), 1274
esp_vfs_fat_sdmmc_unmount (C++ function), 1271
esp_vfs_fat_sdsipi_mount (C++ function), 1271
esp_vfs_fat_spiflash_format_rw_wl
(C++ function), 1273
esp_vfs_fat_spiflash_mount_ro(C++ function), 1273
esp_vfs_fat_spiflash_mount_rw_wl (C++ function), 1272
esp_vfs_fat_spiflash_unmount_ro (C++ function), 1273
esp_vfs_fat_spiflash_unmount_rw_wl
(C++ function), 1272
esp_vfs_fat_unregister_path (C++ function), 1270
ESP_VFS_FLAG_CONTEXT_PTR (C macro), 1348
ESP_VFS_FLAG_DEFAULT (C macro), 1348
ESP_VFS_FLAG_READONLY_FS (C macro), 1349
esp_vfs_fstat (C++ function), 1340
esp_vfs_id_t (C++ type), 1349
esp_vfs_l2tap_eth_filter (C++ function), 433
esp_vfs_l2tap_intf_register (C++ function), 433
esp_vfs_t::truncate (C++ member), 1347
esp_vfs_t::truncate_p (C++ member), 1347
esp_vfs_t::unlink (C++ member), 1345
esp_vfs_t::unlink_p (C++ member), 1345
esp_vfs_t::utime (C++ member), 1347
esp_vfs_t::utime_p (C++ member), 1347
esp_vfs_t::write (C++ member), 1344
esp_vfs_t::write_p (C++ member), 1344
esp_vfs_unlink (C++ function), 1340
esp_vfs_unregister (C++ function), 1341
esp_vfs_unregister_fd (C++ function), 1342
esp_vfs_unregister_with_id (C++ function), 1341
esp_vfs_usb_serial_jtag_use_driver (C++ function), 1351
esp_vfs_usb_serial_jtag_use_nonblocking (C++ function), 1351
esp_vfs_utime (C++ function), 1340
esp_vfs_write (C++ function), 1340
esp_wifi_80211_tx (C++ function), 1702
esp_wifi_ap_get_sta_aid (C++ function), 281
esp_wifi_ap_get_sta_list (C++ function), 280
esp_wifi_ap_wps_disable (C++ function), 343
esp_wifi_ap_wps_enable (C++ function), 342
esp_wifi_ap_wps_start (C++ function), 343
esp_wifi_clear_ap_list (C++ function), 275
esp_wifi_clear_default_wifi_driver_and_handlers (C++ function), 435
esp_wifi_clear_fast_connect (C++ function), 273
esp_wifi_config_11b_rate (C++ function), 286
esp_wifi_config_80211_tx_rate (C++ function), 288
esp_wifi_config_esnnow_rate (C++ function), 223
esp_wifi_connect (C++ function), 273
ESP_WIFI_CONNECTIONLESS_INTERVAL_DEFAULT (C macro), 293
esp_wifi_connectionless_module_set_wake (C++ function), 286
esp_wifi_deinit (C++ function), 274
esp_wifi_deinit (C++ function), 288
esp_wifi_disconnect (C++ function), 273
esp_wifi_force_wakeup_acquire (C++ function), 286
esp_wifi_force_wakeup_release (C++ function), 287
esp_wifi_ftm_end_session (C++ function), 286
esp_wifi_ftm_initiate_session (C++ function), 285
esp_wifi_ftm_resp_set_offset (C++ function), 286
esp_wifi_get_ant (C++ function), 284
esp_wifi_get_ant_gpio (C++ function), 284
esp_wifi_get.bandwidth (C++ function), 276
esp_wifi_get_channel (C++ function), 277
esp_wifi_get_config (C++ function), 280
esp_wifi_get.country (C++ function), 278
esp_wifi_get.country_code (C++ function), 287
esp_wifi_get_event_mask (C++ function), 282
esp_wifi_get.inactive_time (C++ function), 285
esp_wifi_get.mac (C++ function), 278
esp_wifi_get_max_tx_power (C++ function), 282
esp_wifi_get_mode (C++ function), 272
esp_wifi_get_promiscuous (C++ function), 279
esp_wifi_get_promiscuous_ctrl_filter (C++ function), 279
esp_wifi_get_promiscuous_filter (C++ function), 279
esp_wifi_get_protocol (C++ function), 276
esp_wifi_get_ps (C++ function), 275
esp_wifi_get_tsf_time (C++ function), 284
esp_wifi_init (C++ function), 271
ESP_WIFI_MAX_CONN_NUM (C macro), 319
ESP_WIFI_MAX_FILTER_LEN (C macro), 321
ESP_WIFI_MAX_SVC_INFO_LEN (C macro), 321
esp_wifi_nan_cancel_service (C++ function), 353
esp_wifi_nan_datapath_end (C++ function), 354
ESP_WIFI_NAN_DATAPATH_MAX_PEERS (C macro), 321
esp_wifi_nan_datapath_req (C++ function), 353
esp_wifi_nan_datapath_resp (C++ function), 354
esp_wifi_nan_get_ipv6_linklocal_from_mac (C++ function), 354
esp_wifi_nan_get_own_svc_info (C++ function), 354
esp_wifi纳米 get peer info (C++ function), 355
esp_wifi纳米 get peer_records (C++ function), 354
ESP_WIFI_NAN_MAX_SVC_SUPPORTED (C macro), 320
esp_wifi纳米 publish service (C++ function), 353
esp_wifi纳米 send message (C++ function), 353
esp_wifi纳米 start (C++ function), 352
esp_wifi纳米 stop (C++ function), 353
esp_wifi纳米 subscribe service (C++ function), 353
ESP_WIFI_NDP_ROLE_INITIATOR (C macro),
Index

esson_spi_rddma_seg (C++ function), 168
esson_spi_read_reg (C++ function), 165
esson_spi_reset_cnt (C++ function), 166
esson_spi_send_packet (C++ function), 166
esson_spi_wrbuf (C++ function), 167
esson_spi_wrbuf_pulling (C++ function), 168
esson_spi_wrdma (C++ function), 169
esson_spi_wrdma_done (C++ function), 170
esson_spi_wrdma_seg (C++ function), 169
esson_spi_write_reg (C++ function), 166
es_task_state (C++ function), 1453
eTaskState (C++ enum), 1475
eTaskState::eBlocked (C++ enumerator), 1475
eTaskState::eDeleted (C++ enumerator), 1475
eTaskState::eInvalid (C++ enumerator), 1475
eTaskState::eReady (C++ enumerator), 1475
eTaskState::eRunning (C++ enumerator), 1475
eTaskState::eSuspended (C++ enumerator), 1475
ETH_DEFAULT_CONFIG (C macro), 368
ETH_DEFAULT_SPI (C macro), 378
ether_event_t (C++ enum), 371
ether_event_t::ETHERNET_EVENT_CONNECTED (C++ enumerator), 372
ether_event_t::ETHERNET_EVENT_DISCONNECTED (C++ enumerator), 372
ether_event_t::ETHERNET_EVENT_STOP (C++ enumerator), 371
ether_mac_clock_config_t (C++ union), 372
ether_mac_clock_config_t::clock_gpio (C++ member), 372
ether_mac_clock_config_t::clock_mode (C++ member), 372
ether_mac_clock_config_t::mi (C++ member), 372
ether_mac_clock_config_t::rmii (C++ member), 372
ether_mac_config_t (C++ struct), 377
ether_mac_config_t::flags (C++ member), 377
ether_mac_config_t::rx_task_prio (C++ member), 377
ether_mac_config_t::rx_task_stack_size (C++ member), 377
ether_mac_config_t::sw_reset_timeout_ms (C++ member), 377
ETH_MAC_DEFAULT_CONFIG (C macro), 378
ETH_MAC_FLAG_PIN_TO_CORE (C macro), 378
ETH_MAC_FLAG_WORK_WITH_CACHE_DISABLE (C macro), 378
ether PHY autoneg_cmd_t (C++ enum), 384
ether PHY autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_DIS (C++ enumerator), 385
ether PHY autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_END (C++ enumerator), 385
ether PHY autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_G_STAT (C++ enumerator), 385
ether PHY autoneg_cmd_t::ESP_ETH_PHY_AUTONEGO_RESTART (C++ enumerator), 384
ether PHY_config_t (C++ struct), 384
ether PHY_config_t::autoneg_timeout_ms (C++ member), 384
ether PHY_config_t::phy_addr (C++ member), 384
ether PHY_config_t::reset_gpio_num (C++ member), 384
ether PHY_config_t::reset_timeout_ms (C++ member), 384
ETH_PHY_DEFAULT_CONFIG (C macro), 384
ether_SPI_custom_driver_config_t (C++ struct), 377
ether_SPI_custom_driver_config_t::conf (C++ member), 377
ether_SPI_custom_driver_config_t::deinit (C++ member), 377
ether_SPI_custom_driver_config_t::init (C++ member), 377
ether_SPI_custom_driver_config_t::read (C++ member), 378
eventfd (C++ function), 1351
EventGroupHandle_t (C++ type), 1536
ff_diskio_impl_t (C++ struct), 1267
ff_diskioImpl_t::init (C++ member), 1267
ff_diskioImpl_t::ioctl (C++ member), 1268
ff diskio_impl_t::read (C++ member), 1267
ff_diskio_impl_t::status (C++ member), 1267
ff_diskio_impl_t::write (C++ member), 1267
ff_diskio_register (C++ function), 1267
ff_diskio_register_raw_partition (C++ function), 1268
ff_diskio_register_sdmmc (C++ function), 1268
ff_diskio_register_wl_partition (C++ function), 1268

gpio_glitch_filter_enable (C++ function), 510
gpio_glitch_filter_disable (C++ function), 495
gpio_install_isr_service (C++ function), 494

gpio_int_type_t (C++ enum), 501
gpio_int_type_t::GPIO_INTR_ANYEDGE (C++ enum), 501
gpio_int_type_t::GPIO_INTR_DISABLE (C++ enum), 501
gpio_int_type_t::GPIO_INTR_HIGH_LEVEL (C++ enum), 501
gpio_int_type_t::GPIO_INTR_LOW_LEVEL (C++ enum), 501
gpio_int_type_t::GPIO_INTR_MAX (C++ enum), 501
gpio_int_type_t::GPIO_INTR_NEGEDGE (C++ enum), 501
gpio_int_type_t::GPIO_INTR_POSEDGE (C++ enum), 501
gpio_intr_disable (C++ function), 491
gpio_intr_enable (C++ function), 491
gpio_iomux_in (C++ function), 496
gpio_iomux_out (C++ function), 496
GPIO_IS_VALID_DIGITAL_IO_PAD (C macro), 498
GPIO_IS_VALID_GPIO (C macro), 498
GPIO_IS_VALID_OUTPUT_GPIO (C macro), 498
GPIO_IS_VALID_OUTPUT_OD (C macro), 498
GPIO_IS_VALID_INPUT_GPIO (C macro), 498
GPIO_IS_VALID_INPUT (C macro), 498

Releasemaster
GPIO_PIN_REG_0 (C macro), 498
GPIO_PIN_REG_1 (C macro), 498
GPIO_PIN_REG_10 (C macro), 499
GPIO_PIN_REG_11 (C macro), 499
GPIO_PIN_REG_12 (C macro), 499
GPIO_PIN_REG_13 (C macro), 499
GPIO_PIN_REG_14 (C macro), 499
GPIO_PIN_REG_15 (C macro), 499
GPIO_PIN_REG_16 (C macro), 499
GPIO_PIN_REG_17 (C macro), 499
GPIO_PIN_REG_18 (C macro), 499
GPIO_PIN_REG_19 (C macro), 499
GPIO_PIN_REG_2 (C macro), 498
GPIO_PIN_REG_20 (C macro), 499
GPIO_PIN_REG_21 (C macro), 499
GPIO_PIN_REG_22 (C macro), 499
GPIO_PIN_REG_23 (C macro), 499
GPIO_PIN_REG_24 (C macro), 499
GPIO_PIN_REG_25 (C macro), 499
GPIO_PIN_REG_26 (C macro), 499
GPIO_PIN_REG_27 (C macro), 499
GPIO_PIN_REG_28 (C macro), 499
GPIO_PIN_REG_29 (C macro), 500
GPIO_PIN_REG_3 (C macro), 498
GPIO_PIN_REG_30 (C macro), 500
GPIO_PIN_REG_31 (C macro), 500
GPIO_PIN_REG_32 (C macro), 500
GPIO_PIN_REG_33 (C macro), 500
GPIO_PIN_REG_34 (C macro), 500
GPIO_PIN_REG_35 (C macro), 500
GPIO_PIN_REG_36 (C macro), 500
GPIO_PIN_REG_37 (C macro), 500
GPIO_PIN_REG_38 (C macro), 500
GPIO_PIN_REG_39 (C macro), 500
GPIO_PIN_REG_4 (C macro), 498
GPIO_PIN_REG_40 (C macro), 500
GPIO_PIN_REG_41 (C macro), 500
GPIO_PIN_REG_42 (C macro), 500
GPIO_PIN_REG_43 (C macro), 500
GPIO_PIN_REG_44 (C macro), 500
GPIO_PIN_REG_45 (C macro), 500
GPIO_PIN_REG_46 (C macro), 500
GPIO_PIN_REG_47 (C macro), 500
GPIO_PIN_REG_48 (C macro), 500
GPIO_PIN_REG_49 (C macro), 500
GPIO_PIN_REG_5 (C macro), 498
GPIO_PIN_REG_50 (C macro), 500
GPIO_PIN_REG_51 (C macro), 500
GPIO_PIN_REG_52 (C macro), 501
GPIO_PIN_REG_53 (C macro), 501
GPIO_PIN_REG_54 (C macro), 501
GPIO_PIN_REG_55 (C macro), 501
GPIO_PIN_REG_56 (C macro), 501
GPIO_PIN_REG_6 (C macro), 499
GPIO_PIN_REG_7 (C macro), 499
GPIO_PIN_REG_8 (C macro), 499
GPIO_PIN_REG_9 (C macro), 499
gpio_port_t (C++ enum), 501
gpio_port_t::GPIO_PORT_0 (C++ enumerator), 501
gpio_port_t::GPIO_PORT_MAX (C++ enumerator), 501
gpio_pull_mode_t (C++ enum), 502
gpio_pull_mode_t::GPIO_FLOATING (C++ enumerator), 502
gpio_pull_mode_t::GPIO_PULLDOWN_ONLY (C++ enumerator), 502
gpio_pull_mode_t::GPIO_PULLUP_ONLY (C++ enumerator), 502
gpio_pull_mode_t::GPIO_PULLUP_PULLDOWN (C++ enumerator), 502
gpio_pulldown_dis (C++ function), 494
gpio_pulldown_en (C++ function), 494
gpio_pulldown_t (C++ enum), 502
gpio_pulldown_t::GPIO_PULLDOWN_DISABLE (C++ enumerator), 502
gpio_pulldown_t::GPIO_PULLDOWN_ENABLE (C++ enumerator), 502
gpio_pullup_dis (C++ function), 493
gpio_pullup_en (C++ function), 493
gpio_pullup_t (C++ enum), 502
gpio_pullup_t::GPIO_PULLUP_DISABLE (C++ enumerator), 502
gpio_pullup_t::GPIO_PULLUP_ENABLE (C++ enumerator), 502
gpio_reset_pin (C++ function), 491
gpio_set_direction (C++ function), 492
gpio_set_drive_capability (C++ function), 495
gpio_set_intr_type (C++ function), 491
gpio_set_level (C++ function), 492
gpio_set_pull_mode (C++ function), 492
gpio_sleep_sel_dis (C++ function), 496
gpio_sleep_sel_en (C++ function), 496
gpio_sleep_set_direction (C++ function), 497
gpio_sleep_set_pull_mode (C++ function), 497
gpio_uninstall_isr_service (C++ function), 494
gpio_wakeup_disable (C++ function), 493
gpio_wakeup_enable (C++ function), 493
gptimer_alarm_cb_t (C++ type), 524
gptimer_alarm_config_t (C++ struct), 521
gptimer_alarm_config_t::alarm_count (C++ member), 522
gptimer_alarm_config_t::auto_reload_on_alarm (C++ member), 522
gptimer_alarm_config_t::flags (C++ member), 522
gptimer_alarm_config_t::reload_count (C++ member), 522
gptimer_alarm_event_data_t (C++ struct), 523
gptimer_alarm_event_data_t::alarm_value (C++ member), 523
gptimer_alarm_event_data_t::count_value (C++ member), 523

gptimer_clock_source_t (C++ type), 524

gptimer_config_t (C++ struct), 521

gptimer_config_t::clk_src (C++ member), 521

gptimer_config_t::direction (C++ member), 521

gptimer_config_t::flags (C++ member), 521

gptimer_config_t::intr_priority (C++ member), 521

gptimer_config_t::intr_shared (C++ member), 521

gptimer_config_t::resolution_hz (C++ member), 521

gptimer_count_direction_t (C++ enum), 524

gptimer_count_direction_t::GPTIMER_COUNT_DOWN (C++ enumerator), 524

gptimer_del_timer (C++ function), 516

gptimer_disable (C++ function), 519

gptimer_enable (C++ function), 519

gptimer_etm_event_config_t (C++ struct), 523

gptimer_etm_event_config_t::event_type (C++ member), 523

gptimer_etm_event_type_t (C++ enum), 525

gptimer_etm_event_type_t::GPTIMER_ETM_ALARM_MATCH (C++ enumerator), 525

gptimer_etm_event_type_t::GPTIMER_ETM_EVENT_MAX (C++ function), 525

gptimer_etm_task_config_t (C++ struct), 523

gptimer_etm_task_config_t::task_type (C++ member), 523

gptimer_etm_task_type_t (C++ enum), 524

gptimer_etm_task_type_t::GPTIMER_ETM_TASK_CAPTURE (C++ enumerator), 525

gptimer_etm_task_type_t::GPTIMER_ETM_TASK_MAX (C++ function), 525

gptimer_event_callbacks_t (C++ struct), 521

gptimer_event_callbacks_t::on_alarm (C++ member), 521

gptimer_get_captured_count (C++ function), 518

gptimer_get_raw_count (C++ function), 517

gptimer_get_resolution (C++ function), 517

gptimer_handle_t (C++ type), 524

gptimer_new_etm_event (C++ function), 522

gptimer_new_etm_task (C++ function), 522

gptimer_new_timer (C++ function), 516

gptimer_register_event_callbacks (C++ function), 518

gptimer_set_alarm_action (C++ function), 519

gptimer_set_raw_count (C++ function), 516

gptimer_start (C++ function), 520

gptimer_stop (C++ function), 520

heap_caps_add_region (C++ function), 1591
heap_caps_add_region_with_caps (C++ function), 1592
heap_caps_aligned_alloc (C++ function), 1585
heap_caps_aligned_malloc (C++ function), 1585
heap_caps_aligned_free (C++ function), 1586
heap_caps_arm_alloc (C++ function), 1586
heap_caps_check_integrity (C++ function), 1587
heap_caps_check_integrity_addr (C++ function), 1588
heap_caps_check_integrity_all (C++ function), 1587
heap_caps_check_integrity_all (C++ function), 1589
heap_caps_compress_block (C++ function), 1590
heap_caps_no_nonos_stack_heaps (C++ function), 1591
heap_caps_free (C++ function), 1585
heap_caps_get_allocated_size (C++ function), 1589
heap_caps_get_free_size (C++ function), 1586
heap_caps_get_info (C++ function), 1587
heap_caps_get_largest_free_block (C++ function), 1587
heap_caps_get_minimum_free_size (C++ function), 1586
heap_caps_get_total_size (C++ function), 1586
heap_caps_init (C++ function), 1591
heap_caps_malloc (C++ function), 1585
heap_caps_malloc_extmem_enable (C++ function), 1588
heap_caps_malloc_prefer (C++ function), 1588
heap_caps_malloc_prefer (C++ function), 1585
HEAP_IRAM_ATTR (C macro), 1589
heap_trace_dump (C++ function), 1618
heap_trace_dump_caps (C++ function), 1618
heap_trace_get (C++ function), 1618
heap_trace_get_count (C++ function), 1618
heap_trace_init_standalone (C++ function), 1617
heap_trace_init_tohost (C++ function), 1617
heap_trace_mode_t (C++ enum), 1620
heap_trace_mode_t::HEAP_TRACE_ALL (C++ enumerator), 1620
heap_trace_mode_t::HEAP_TRACE_LEAKS (C++ enumerator), 1620
heap_trace_record_t (C++ struct), 1619
heap_trace_record_t (C++ type), 1620
heap_trace_record_t::address (C++ member), 1619
heap_trace_record_t::allocated_by (C++ member), 1619
heap_trace_record_t::count (C++ member), 1619
heap_trace_record_t::freed_by (C++ member), 1619
heap_trace_record_t::size (C++ member), 1619
heap_trace_resume (C++ function), 1618
heap_trace_start (C++ function), 1617
heap_trace_stop (C++ function), 1617
heap_trace_summary (C++ function), 1618
heap_trace_summary_t (C++ struct), 1619
heap_trace_summary_t::capacity (C++ member), 1619
heap_trace_summary_t::count (C++ member), 1619
heap_trace_summary_t::has_overflowed (C++ member), 1619
heap_trace_summary_t::high_water_mark (C++ member), 1619
heap_trace_summary_t::mode (C++ member), 1619
heap_trace_summary_t::total_allocations (C++ member), 1619
heap_trace_summary_t::total_frees (C++ member), 1619
hmac_key_id_t (C++ enum), 535
hmac_key_id_t::HMAC_KEY0 (C++ enumerator), 535
hmac_key_id_t::HMAC_KEY1 (C++ enumerator), 535
hmac_key_id_t::HMAC_KEY2 (C++ enumerator), 535
hmac_key_id_t::HMAC_KEY3 (C++ enumerator), 535
hmac_key_id_t::HMAC_KEY4 (C++ enumerator), 535
hmac_key_id_t::HMAC_KEY5 (C++ enumerator), 535
hmac_key_id_t::HMAC_KEY_MAX (C++ enumerator), 535
http_client_init_cb_t (C++ type), 1422
http_event_handle_cb (C++ type), 143
HTTPD_200 (C macro), 196
HTTPD_204 (C macro), 196
HTTPD_207 (C macro), 196
HTTPD_400 (C macro), 196
HTTPD_404 (C macro), 196
HTTPD_408 (C macro), 196
HTTPD_500 (C macro), 196
httpd_close_func_t (C++ type), 199
httpd_config (C++ struct), 192
httpd_config::backlog_conn (C++ member), 193
httpd_config::close_fn (C++ member), 192
httpd_config::core_id (C++ member), 192
httpd_config::ctrl_port (C++ member), 192
httpd_config::enable_so_linger (C++ member), 193
httpd_config::global_transport_ctx (C++ member), 193
httpd_config::global_transport_ctx_free_fn (C++ member), 193
httpd_config::global_user_ctx (C++ member), 193
httpd_config::global_user_ctx_free_fn (C++ member), 193
httpd_config::keep_alive_count (C++ member), 194
httpd_config::keep_alive_enable (C++ member), 193
httpd_config::keep_alive_id (C++ member), 194
httpd_config::keep_alive_interval (C++ member), 194
httpd_config::linger_timeout (C++ member), 193
httpd_config::lru_purge_enable (C++ member), 193
httpd_config::max_open_sockets (C++ member), 193
httpd_config::max_resp_headers (C++ member), 193
httpd_config::max_uri_handlers (C++ member), 193
httpd_config::open_fn (C++ member), 194
httpd_config::recv_wait_timeout (C++ member), 193
httpd_config::send_wait_timeout (C++ member), 193
httpd_config::server_port (C++ member), 192
httpd_config::task_priority (C++ member), 192
httpd_config::uri_match_fn (C++ member), 194
httpd_config_t (C++ type), 200
HTTPD_DEFAULT_CONFIG (C macro), 197
httpd_ssl_config::cacert_pem (C++ member), 203
httpd_ssl_config::ecdsa_key_efuse_blk (C++ member), 204
httpd_ssl_config::httpd (C++ member), 203
httpd_ssl_config::port_insecure (C++ member), 204
httpd_ssl_config::port_secure (C++ member), 204
httpd_ssl_config::prvkey_len (C++ member), 204
httpd_ssl_config::prvkey_pem (C++ member), 204
httpd_ssl_config::servername (C++ member), 203
httpd_ssl_config::session_tickets (C++ member), 204
httpd_ssl_config::ssl_userdata (C++ member), 204
httpd_ssl_config::transport_mode (C++ member), 204
httpd_ssl_config::use_ecdsa_peripheral (C++ member), 204
HTTPD_SSL_CONFIG_DEFAULT (C macro), 204
httpd_ssl_config_default (C++ function), 202
httpd_ssl_config_t (C++ type), 205
httpd_ssl_start (C++ function), 203
httpd_ssl_stop (C++ function), 203
httpd_ssl_transport_mode_t (C++ enum), 205
httpd_ssl_transport_mode_t::HTTPD_SSL_TRANSPORT_INSECURE (C++ enumerator), 205
httpd_ssl_transport_mode_t::HTTPD_SSL_TRANSPORT_SECURE (C++ enumerator), 205
httpd_ssl_user_cb_state_t (C++ enum), 205
httpd_ssl_user_cb_state_t::HTTPD_SSL_USER_CB_STATE.Create (C++ enumerator), 205
httpd_ssl_user_cb_state_t::HTTPD_SSL_USER_CB_STATE.Delete (C++ enumerator), 205
httpd_ssl_user_cb_state_t::HTTPD_SSL_USER_CB_STATE.Update (C++ enumerator), 205
http_start (C++ function), 188
http_stop (C++ function), 189
HTTPD_TYPE_JSON (C macro), 196
HTTPD_TYPE_OCTET (C macro), 196
HTTPD_TYPE_TEXT (C macro), 196
http_unregister_uri (C++ function), 177
http_unregister_uri_handler (C++ function), 177
http_uri (C++ struct), 195
http_uri::handler (C++ member), 196
http_uri::method (C++ member), 195
http_uri::uri (C++ member), 195
httpd_uri::user_ctx (C++ member), 196
httpd_uri_match_func_t (C++ type), 199
httpd_uri_matchWildcard (C++ function), 182
httpd_uri_t (C++ type), 197
httpd_work_fn_t (C++ type), 200
HttpStatus_Code::HttpStatus_BadRequest (C++ enumerator), 145
HttpStatus_Code::HttpStatus_BadRequest (C++ enumerator), 146
HttpStatus_Code::HttpStatus_Forbidden (C++ enumerator), 146
HttpStatus_Code::HttpStatus_Forbidden (C++ enumerator), 146
HttpStatus_Code::HttpStatus_InternalError (C++ enumerator), 146
HttpStatus_Code::HttpStatus_MovedPermanently (C++ enumerator), 146
HttpStatus_Code::HttpStatus_MultipleChoices (C++ enumerator), 145
HttpStatus_Code::HttpStatus_NotFound (C++ enumerator), 146
HttpStatus_Code::HttpStatus_Ok (C++ enumerator), 145
HttpStatus_Code::HttpStatus_SeeOther (C++ enumerator), 146
HttpStatus_Code::HttpStatus_TemporaryRedirect (C++ enumerator), 146
HttpStatus_Code::HttpStatus_TemporaryRedirect (C++ enumerator), 146
HttpStatus_Code::HttpStatusUnauthorized (C++ enumerator), 146

i2c_ack_type_t (C++ enum), 562
i2c_ack_type_t::I2C_MASTER_ACK (C++ enumerator), 562
i2c_ack_type_t::I2C_MASTER_ACK_MAX (C++ enumerator), 562
i2c_ack_type_t::I2C_MASTER_LAST_NACK (C++ enumerator), 562
i2c_ack_type_t::I2C_MASTER_NACK (C++ enumerator), 562
i2c_addr_bit_len_t::I2C_ADDR_BIT_LEN_7 (C++ enum), 561
i2c_addr_bit_len_t::I2C_ADDR_BIT_LEN_8 (C++ enum), 561
i2c_addr_bit_len_t::I2C_ADDR_BIT_LEN_9 (C++ enum), 561
i2c_addr_bit_len_t::I2C_ADDR_BIT_LEN_10 (C++ enum), 562
i2c_addr_bit_len_t::I2C_ADDR_BIT_LEN_11 (C++ enum), 562
i2c_addr_bit_len_t::I2C_ADDR_BIT_MAX (C++ enumerator), 562
i2c_addr_mode_t (C++ enum), 562
i2c_addr_mode_t::I2C_ADDR_BIT_10 (C++ enumerator), 562
i2c_addr_mode_t::I2C_ADDR_BIT_7 (C++ enumerator), 562
i2c_addr_mode_t::I2C_ADDR_BIT_MAX (C++ enumerator), 562
i2c_clock_source_t (C++ type), 561
i2c_del_master_bus (C++ function), 551
i2c_del_slave_device (C++ function), 555
i2c_device_config_t (C++ struct), 554
i2c_device_config_t::dev_addr_length (C++ member), 554

Espressif Systems 2267
Submit Document Feedback

Release master
i2c_device_config_t::device_address
(C++ member), 554
i2c_device_config_t::scl_speed_hz
(C++ member), 554
i2c_hal_clk_config_t (C++ struct), 560
i2c_hal_clk_config_t::clkm_div (C++
member), 560
i2c_hal_clk_config_t::hold (C++ member), 561
i2c_hal_clk_config_t::scl_high
(C++ member), 560
i2c_hal_clk_config_t::scl_low
(C++ member), 560
i2c_hal_clk_config_t::scl_wait_high
(C++ member), 560
i2c_hal_clk_config_t::sda_hold
(C++ member), 560
i2c_hal_clk_config_t::sda_sample
(C++ member), 560
i2c_hal_clk_config_t::setup (C++ mem-
ber), 561
i2c_hal_clk_config_t::tout (C++ member), 561
i2c_master_bus_add_device (C++ function), 551
i2c_master_bus_config_t (C++ struct), 554
i2c_master_bus_config_t::clk_source
(C++ member), 554
i2c_master_bus_config_t::enable_internal_pullup
(C++ member), 554
i2c_master_bus_config_t::flags
(C++ member), 554
i2c_master_bus_config_t::glitch_ignore
i2c_master_bus_config_t::i2c_port
(C++ member), 554
i2c_master_bus_config_t::intr_priority
(C++ member), 554
i2c_master_bus_config_t::scl_io_num
(C++ member), 554
i2c_master_bus_config_t::sda_io_num
(C++ member), 554
i2c_master_bus_config_t::trans_queue_depth
(C++ member), 554
i2c_master_bus_handle_t (C++ type), 559
i2c_master_bus_reset (C++ function), 553
i2c_master_bus_rm_device (C++ function), 551
i2c_master_bus_wait_all_done (C++ func-
tion), 553
i2c_master_callback_t (C++ type), 559
i2c_master_event/callbacks_t (C++ struct), 555
i2c_master_event/callbacks_t::on_trans_done
(C++ member), 555
i2c_master_event_data_t (C++ struct), 558
i2c_master_event_data_t::event
(C++ member), 558
i2c_master_event_t (C++ enum), 560
i2c_master_event_t::I2C_EVENT_ALIVE
(C++ enumerator), 560
i2c_master_event_t::I2C_EVENT_DONE
(C++ enumerator), 560
i2c_master_event_t::I2C_EVENT_NACK
(C++ enumerator), 560
i2c_master_event_t::i2c_master_event_data_t::event
(C++ member), 558
i2c_master_event_t::I2C_STATUS_ACK_ERROR
(C++ enumerator), 559
i2c_master_event_t::I2C_STATUSDONE
(C++ enumerator), 560
i2c_master_event_t::I2C_STATUS_IDLE
(C++ enumerator), 559
i2c_master_event_t::I2C_STATUS_READ
(C++ enumerator), 559
i2c_master_event_t::I2C_STATUS_START
(C++ enumerator), 559
i2c_master_event_t::I2C_STATUS_STOP
(C++ enumerator), 559
i2c_master_event_t::I2C_STATUS_TIMEOUT
(C++ enumerator), 560
i2c_master_event_t::I2C_STATUS_WRITE
(C++ enumerator), 559
i2c_master_event_t::i2c_port_num
(C++ function), 551
i2c_master_transmit_recv (C++ function), 551
i2c_master_transmit (C++ function), 551
i2c_mode_t (C++ enum), 561
i2c_mode_t::I2C_MODE_MASTER (C++ enum-
ator), 561
i2c_mode_t::I2C_MODE_MAX (C++ enum-
ator), 561
i2c_mode_t::I2C_MODE_SLAVE (C++ enum-
ator), 561
i2c_new_master_bus (C++ function), 550
i2c_new_slave_device (C++ function), 555
i2c_port_num_t (C++ type), 559
i2c_port_t (C++ enum), 561
i2c_port_t::I2C_NUM_0 (C++ enumerator), 561
i2c_port_t::I2C_NUM_1 (C++ enumerator), 561
i2c_port_t::I2C_NUM_MAX (C++ enumer-
ator), 561
i2c_rw_t (C++ enum), 562
i2c_rw_t::I2C_MASTER_WRITE (C++ enum-
ator), 562
i2c_slave_config_t (C++ struct), 557
i2c_slave_config_t::addr_bit_len
(C++ member), 557
i2c_slave_config_t::clk_source
(C++ member), 557
i2c_slave_config_t::flags
(C++ member), 557
i2c_slave_config_t::intr_priority
(C++ member), 557
i2c_slave_config_t::sda_io_num
(C++ member), 558
i2c_slave_config_t::scl_speed_hz
(C++ member), 558
i2s_std_gpio_config_t::bclk_inv (C++ member), 577
i2s_std_gpio_config_t::din (C++ member), 577
i2s_std_gpio_config_t::dout (C++ member), 577
i2s_std_gpio_config_t::invert_flags (C++ member), 577
i2s_std_gpio_config_t::mclk (C++ member), 577
i2s_std_gpio_config_t::mclk_inv (C++ member), 577
i2s_std_config_t::ws (C++ member), 577
i2s_std_config_t::ws_inv (C++ member), 577
I2S_STD_MSB_SLOT_DEFAULT_CONFIG (C macro), 578
I2S_STD_PCM_SLOT_DEFAULT_CONFIG (C macro), 578
I2S_STD_PHILIPS_SLOT_DEFAULT_CONFIG (C macro), 578
i2s_slot_bit_width_t::bit_shift (C++ member), 576
i2s_slot_config_t::msb_right (C++ member), 576
i2s_slot_config_t::slot_mask (C++ member), 576
i2s_slot_config_t::ws_width (C++ member), 576
i2s_slot_mode_t::ws (C++ member), 577
i2s_slot_mode_t::ws_inv (C++ member), 577
i2s_slot_mode_t::ws_width (C++ member), 577
i2s_slot_mask_t::I2S_STD_SLOT_DEFAULT_CONFIG (C++ enumerato,
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2TAP_VFS_DEFAULT_PATH</td>
<td>C macro</td>
</tr>
<tr>
<td>l2tap_vfs_config_t</td>
<td>C struct</td>
</tr>
<tr>
<td>L2TAP_VFS_CONFIG_DEFAULT</td>
<td>C macro</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t</td>
<td>C struct</td>
</tr>
<tr>
<td>L2TAP_S_RCV_FILTER</td>
<td>C macro</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_S_RCV_FILTER</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_RCV_FILTER</td>
<td>C macro</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_RCV_FILTER</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_INTF_DEVICE</td>
<td>C macro</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_INTF_DEVICE</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL</td>
<td>C struct</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::index</td>
<td>C enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::index</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev</td>
<td>C struct</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::index</td>
<td>C enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::index</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::name</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::name</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::index</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::index</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::name</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::name</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::index</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::index</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::name</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::name</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::index</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::index</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::name</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::name</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value::index</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value::index</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value::name</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value::name</td>
<td>C member</td>
</tr>
<tr>
<td>L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value::value</td>
<td>C Enum</td>
</tr>
<tr>
<td>l2tap_ioctl_opt_t::L2TAP_G_DEVICE_DRV_HNDL::dev::prop::value::value::value::value</td>
<td>C member</td>
</tr>
</tbody>
</table>
Index

lcd_color_range_t (C++ enum), 594  ledc_channel_config_t::hpoint (C++ member), 620
lcd_color_range_t::LCD_COLOR_RANGE_FULL  ledc_channel_config_t::intr_type (C++ member), 620
lcd_color_range_t::LCD_COLOR_RANGE_LIMIT  ledc_channel_config_t::output_invert (C++ member), 620
lcd_color_range_t::LCD_COLOR_SPACE_RGB  ledc_channel_config_t::speed_mode (C++ member), 619
lcd_color_range_t::LCD_COLOR_SPACE_YUV  ledc_channel_config_t::timer_sel (C++ member), 620
lcd_rgb_data_endian_t (C++ enum), 593  ledc_channel_t (C++ enum), 624
lcd_rgb_data_endian_t::LCD_RGB_DATA_ENDIAN_LEDC_CHANNEL_0 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_MAX (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_0 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_1 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_2 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_3 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_4 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_5 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_6 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_7 (C++ enumerator), 624
ledc_channel_t::LEDC_CHANNEL_MAX (C++ enumerator), 625
ledc_clk_cfg_t (C++ type), 622
ledc_clk_cfg_t::LED_APB_CLK (C++ enumerator), 623
ledc_clk_cfg_t::LED_SCLK (C++ enumerator), 623
ledc_duty_direction_t (C++ enum), 623
ledc_duty_direction_t::LED_DUTY_DIR_DECREASE (C++ enumerator), 623
ledc_duty_direction_t::LED_DUTY_DIR_INCREASE (C++ enumerator), 623
ledc_duty_direction_t::LED_DUTY_DIR_MAX (C++ enumerator), 623
LED_APB_CLK_HZ (C macro), 621
ledc_bound_channel_timer (C++ function), 615
ledc_cb_event_t (C++ enum), 622
ledc_channel_config_t::EVENT_LED_FADE_END_EVT (C++ enumerator), 622
ledc_channel_config_t::channel (C++ member), 621
ledc_channel_config_t::duty (C++ member), 621
ledc_channel_config_t::speed_mode (C++ member), 621
ledc_channel_register (C++ function), 619
ledc_cb_t (C++ type), 622
ledc_cbs_t (C++ struct), 621
ledc_cbs_t::fade_cb (C++ member), 621
ledc_channel_config (C++ function), 610
ledc_channel_config_t (C++ struct), 619
ledc_channel_config_t::channel (C++ member), 620
ledc_channel_config_t::duty (C++ member), 620
ledc_channel_config_t::flags (C++ member), 620
ledc_channel_config_t::gpio_num (C++ member), 619
ledc_fade_start (C++ function), 617
ledc_fade_stop (C++ function), 617
Index

ledc_find_suitable_duty_resolution
(C++ function), 610
ledc_get_duty (C++ function), 613
ledc_get_freq (C++ function), 612
ledc_get_hpoint (C++ function), 613
ledc_intr_type_t (C++ enum), 622
ledc_intr_type_t::LEDC_INTR_DISABLE
(C++ enumerator), 623
ledc_intr_type_t::LEDC_INTR_FADE_END
(C++ enumerator), 623
ledc_intr_type_t::LEDC_INTR_MAX
(C++ enumerator), 623
ledc_isr_handle_t (C++ type), 622
ledc_isr_register (C++ function), 614
ledc_mode_t (C++ enum), 622
ledc_mode_t::LEDC_LOW_SPEED_MODE
(C++ enumerator), 622
ledc_mode_t::LEDC_SPEED_MODE_MAX
(C++ enumerator), 622
LED REF CLK H2 (C macro), 621
ledc_set_duty (C++ function), 614
ledc_set_duty_and_update (C++ function), 618
ledc_set_duty_with_hpoint (C++ function), 612
ledc_set_fade (C++ function), 614
ledc_set_fade_step_and_start (C++ function), 618
ledc_set_fade_time_and_start (C++ function), 618
ledc_set_fade_with_step (C++ function), 615
ledc_set_freq (C++ function), 612
ledc_set_freq_with_time (C++ function), 616
ledc_set_pin (C++ function), 611
ledc_slow_clk_sel_t (C++ enum), 623
ledc_slow_clk_sel_t::LEDC_SLOW_CLK_APB
(C++ enumerator), 623
ledc_slow_clk_sel_t::LEDC_SLOW_CLK_RC_FAST
(C++ enumerator), 623
ledc_slow_clk_sel_t::LEDC_SLOW_CLK_RTC8M
(C++ enumerator), 623
ledc_slow_clk_sel_t::LEDC_SLOW_CLK_XTAL
(C++ enumerator), 623
ledc_stop (C++ function), 612
ledc_timer_bit_t (C++ enum), 625
ledc_timer_bit_t::LEDC_TIMER_10_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_11_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_12_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_13_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_14_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_1_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_2_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_3_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_4_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_5_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_6_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_7_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_8_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_9_BIT
(C++ enumerator), 625
ledc_timer_bit_t::LEDC_TIMER_BIT_MAX
(C++ enumerator), 625
ledc_timer_config (C++ function), 610
ledc_timer_config_t (C++ struct), 620
ledc_timer_config_t::clk_cfg (C++ member), 620
ledc_timer_config_t::deconfigure (C++ member), 620
ledc_timer_config_t::duty_resolution
(C++ member), 620
ledc_timer_config_t::freq_hz (C++ member), 620
ledc_timer_config_t::speed_mode (C++ member), 620
ledc_timer_config_t::timer_num
(C++ member), 620
ledc_timer_pause (C++ function), 615
ledc_timer_resume (C++ function), 615
ledc_timer_start (C++ function), 615
ledc_timer_set (C++ function), 614
ledc_timer_t (C++ enum), 624
ledc_timer_t::LEDC_TIMER_0
(C++ enumerator), 624
ledc_timer_t::LEDC_TIMER_1
(C++ enumerator), 624
ledc_timer_t::LEDC_TIMER_2
(C++ enumerator), 624
ledc_timer_t::LEDC_TIMER_3
(C++ enumerator), 624
ledc_timer_t::LEDC_TIMER_MAX
(C++ enumerator), 624
ledc_update_duty (C++ function), 611
linenoiseCompletions (C++ function), 1383

M
M BE (C macro), 1765
M BG (C macro), 1765
M BL (C macro), 1765
M BRANCH (C macro), 1765
M BX (C macro), 1765
M BXF (C macro), 1765
M BZ (C macro), 1765
M LABEL (C macro), 1764
Espressif Systems

Submit Document Feedback
mesh_event_root_conflict_t::rssi (C++ member), 255
mesh_event_root_fixed_t (C++ struct), 255
mesh_event_root_fixed_t::is_fixed (C++ member), 255
mesh_event_root_switch_req_t (C++ struct), 254
mesh_event_root_switch_req_t::rc_addr (C++ member), 255
mesh_event_root_switch_req_t::reason (C++ member), 255
mesh_event_router_switch_t (C++ type), 263
mesh_event_routing_table_change_t (C++ struct), 255
mesh_event_routing_table_change_t::rt_size_change (C++ member), 257
mesh_event_scan_done_t (C++ struct), 255
mesh_event_scan_done_t::number (C++ member), 256
mesh_event_toDS_state_t (C++ enum), 267
mesh_event_toDS_state_t::MESH_TODS_REACHABLE (C++ enumerator), 267
mesh_event_toDS_state_t::MESH_TODS_UNREACHABLE (C++ enumerator), 267
mesh_event_vote_started_t (C++ struct), 254
mesh_event_vote_started_t::attempts (C++ member), 254
mesh_event_vote_started_t::rc_addr (C++ member), 254
mesh_event_vote_started_t::reason (C++ member), 254
MESH_INIT_CONFIG_DEFAULT (C macro), 262
MESH_MPS (C macro), 259
MESH_MTU (C macro), 259
MESH_OPT_RECV_DS_ADDR (C macro), 261
MESH_OPT_SEND_GROUP (C macro), 261
mesh_opt_t (C++ struct), 256
mesh_opt_t::len (C++ member), 256
mesh_opt_t::type (C++ member), 256
mesh_opt_t::val (C++ member), 256
mesh_proto_t (C++ enum), 265
mesh_proto_t::MESH_PROTO_AP (C++ enumerator), 265
mesh_proto_t::MESH_PROTO_BIN (C++ enumerator), 265
mesh_proto_t::MESH_PROTO_HTTP (C++ enumerator), 265
mesh_proto_t::MESH_PROTO_JSON (C++ enumerator), 265
mesh_proto_t::MESH_PROTO_MQTT (C++ enumerator), 265
mesh_proto_t::MESH_PROTO_STA (C++ enumerator), 265
MESH_PS_DEVICE_DUTY_DEMAND (C macro), 262
MESH_PS_DEVICE_DUTY_REQUEST (C macro), 262
MESH_PS_NETWORK_DUTY_APPLIED_ENTIRE (C macro), 262
MESH_PS_NETWORK_DUTY_APPLIED_UPLINK (C macro), 262
MESH_PS_NETWORK_DUTY_MASTER (C macro), 262
mesh_rc_config_t (C++ union), 252
mesh_rc_config_t::attempts (C++ member), 253
mesh_rc_config_t::rc_addr (C++ member), 253
MESH_ROOT_LAYER (C macro), 259
mesh_router_t (C++ struct), 257
mesh_router_t::allow_router_switch (C++ member), 257
mesh_router_t::bssid (C++ member), 257
mesh_router_t::password (C++ member), 257
mesh_router_t::ssid (C++ member), 257
mesh_router_t::ssid_len (C++ member), 257
mesh_rx_pending_t (C++ struct), 259
mesh_rx_pending_t::toDS (C++ member), 259
mesh_rx_pending_t::toSelf (C++ member), 259
mesh_tos_t (C++ enum), 266
mesh_tos_t::MESH_TOS_DEF (C++ enumerator), 266
mesh_tos_t::MESH_TOS_E2E (C++ enumerator), 266
mesh_tos_t::MESH_TOS_P2P (C++ enumerator), 266
mesh_tx_pending_t (C++ struct), 258
mesh_tx_pending_t::broadcast (C++ member), 259
mesh_tx_pending_t::mgmt (C++ member), 259
mesh_tx_pending_t::to_child (C++ member), 259
mesh_tx_pending_t::to_child_p2p (C++ member), 259
mesh_tx_pending_t::to_parent (C++ member), 258
mesh_tx_pending_t::to_parent_p2p (C++ member), 258
mesh_type_t (C++ enum), 265
mesh_type_t::MESH_IDLE (C++ enumerator), 265
mesh_type_t::MESH_LEAF (C++ enumerator), 265
mesh_type_t::MESH_NODE (C++ enumerator), 265
mesh_type_t::MESH_ROOT (C++ enumerator), 265
mesh_type_t::MESH_STA (C++ enumerator), 265
mesh_vote_reason_t (C++ enum), 266
mesh_vote_reason_t::MESH_VOTE_REASON_CHILD_INITIATED (C++ enumerator), 266
mesh_vote_reason_t::MESH_VOTE_REASON_ROOT_INITIATED (C++ enumerator), 266
mesh_vote_t (C++ struct), 258
mesh_vote_t::config (C++ member), 258
mesh_vote_t::is_rc_specified (C++ member), 258
MessageBufferHandle_t (C++ type), 1554
mip_t (C++ struct), 253
mip_t::ip4 (C++ member), 253
mip_t::port (C++ member), 253
MQTT_ERROR_TYPE_ESP_TLS (C macro), 105
multi_heap_aligned_alloc (C++ function), 1592
multi_heap_aligned_alloc_offs (C++ function), 1595
multi_heap_check (C++ function), 1593
multi_heap_dump (C++ function), 1594
multi_heap_free (C++ function), 1593
multi_heap_free_size (C++ function), 1594
multi_heap_get_allocated_size (C++ function), 1593
multi_heap_get_info (C++ function), 1595
multi_heap_info_t (C++ type), 1595
multi_heap_info_t::allocated_blocks (C++ member), 1595
multi_heap_info_t::free_bytes (C++ member), 1595
multi_heap_info_t::largest_free_size (C++ function), 1593
multi_heap_minimum_free_size (C++ function), 1594
multi_heap_realloc (C++ function), 1593
multi_heap_register (C++ function), 1593
multi_heap_set_lock (C++ function), 1594

N
name_uuid (C++ struct), 1237
name_uuid::name (C++ member), 1237
name_uuid::uuid (C++ member), 1237
NAN_MAX_PEERS_RECORD (C macro), 355
nan_peer_record (C++ struct), 355
nan_peer_record::ndp_id (C++ member), 355
nan_peer_record::own_svc_id (C++ member), 355
nan_peer_record::peer_nmi (C++ member), 355
nan_peer_record::peer_svc_id (C++ member), 355
nan_peer_record::peer_svc_type (C++ member), 355
NDP_STATUS_ACCEPTED (C macro), 355
NDP_STATUS_REJECTED (C macro), 355
neighbor_rep_request_cb (C++ type), 346
non_pref_chan (C++ struct), 348	non_pref_chan::chan (C++ member), 348	non_pref_chan::oper_class (C++ member), 348	non_pref_chan::preference (C++ member), 348
non_pref_chan::reason (C++ member), 348
non_pref_chan_reason (C++ enum), 348	non_pref_chan_reason::NON_PREF_CHAN_REASON_EXT_INT_INTERFERENCE (C++ enumerator), 348	non_pref_chan_reason::NON_PREF_CHAN_REASON_INT_INTERFERENCE (C++ enumerator), 349
non_pref_chan_reason::NON_PREF_CHAN_REASON_RSSI (C++ enumerator), 348	non_pref_chan_reason::NON_PREF_CHAN_REASON_UNSPECIFIED (C++ enumerator), 348
non_pref_chan_s (C++ struct), 348	non_pref_chan_s::chan (C++ member), 348	non_pref_chan_s::non_pref_chan_num (C++ member), 348
nvs_close (C++ function), 1296
nvs_commit (C++ function), 1296
NVS_DEFAULT_PART_NAME (C macro), 1302
nvs_entry_find (C++ function), 1297
nvs_entry_find_in_handle (C++ function), 1298
nvs_entry_info (C++ function), 1299
nvs_entry_info_t (C++ struct), 1299
nvs_entry_info_t::key (C++ member), 1299
nvs_entry_info_t::namespace_name (C++ member), 1299
nvs_entry_info_t::type (C++ member), 1299	nvs_entry_next (C++ function), 1299
nvs_erase_all (C++ function), 1296
nvs_erase_key (C++ function), 1295
nvs_flash_deinit (C++ function), 1286
nvs_flash_deinit_partition (C++ function), 1286	nvs_flash_erase (C++ function), 1287
nvs_flash_erase_partition (C++ function), 1287
nvs_flash_erase_partition_ptr (C++ function), 1287
nvs_flash_generate_keys (C++ function), 1288
nvs_flash_generate_keys_t (C++ type), 1290
nvs_flash_generate_keys_v2 (C++ function), 1289
nvs_flash_get_default_security_scheme
Espressif Systems

Submit Document Feedback

Index

(C++ function), 1289
nvs_flash_init (C++ function), 1285
nvs_flash_init_partition (C++ function), 1286
nvs_flash_init_partition_ptr (C++ function), 1286
nvs_flash_read_cfg_t (C++ type), 1290
nvs_flash_read_security_cfg (C++ function), 1288
nvs_flash_read_security_cfg_v2 (C++ function), 1289
nvs_flash_register_security_scheme (C++ function), 1288
nvs_flash_secure_init (C++ function), 1287
nvs_flash_secure_init_partition (C++ function), 1288
nvs_get_blob (C++ function), 1294
nvs_get_i16 (C++ function), 1292
nvs_get_i32 (C++ function), 1293
nvs_get_i64 (C++ function), 1293
nvs_get_i8 (C++ function), 1292
nvs_get_stats (C++ function), 1296
nvs_get_str (C++ function), 1293
nvs_get_u16 (C++ function), 1292
nvs_get_u32 (C++ function), 1293
nvs_get_u64 (C++ function), 1293
nvs_get_u8 (C++ function), 1292
nvs_get_used_entry_count (C++ function), 1297
nvs_handle (C++ type), 1302
nvs_handle_t (C++ type), 1302
nvs_iterator_t (C++ type), 1302
NVS_KEY_NAME_MAX_SIZE (C macro), 1302
NVS_KEY_SIZE (C macro), 1290
NVS_NS_NAME_MAX_SIZE (C macro), 1302
nvs_open (C++ function), 1294
nvs_open_from_partition (C++ function), 1294
nvs_open_mode (C++ type), 1302
nvs_open_mode_t (C++ type), 1302
nvs_open_mode_t::NV$READONLY (C++ enumerator), 1302
nvs_open_mode_t::NV$READWRITE (C++ enumerator), 1302
NVS_PART_NAME_MAX_SIZE (C macro), 1302
nvs_release_iterator (C++ function), 1299
nvs_sec_cfg_t (C++ struct), 1289
nvs_sec_cfg_t::eky (C++ member), 1289
nvs_sec_cfg_t::tky (C++ member), 1289
nvs_sec_config_flash_enc_t (C++ struct), 1308
nvs_sec_config_flash_enc_t::nnv$keys_part (C++ member), 1308
nvs_sec_config_hmac_t (C++ struct), 1309
nvs_sec_config_hmac_t::hmac_key_id (C++ member), 1309
NVS_SEC_PROVIDER_CFG_FLASH_ENC_DEFAULT (C macro), 1309
nvs_sec_provider_deregister (C++ function), 1308
nvs_sec_provider_register_flash_enc (C++ function), 1308
nvs_sec_provider_register_hmac (C++ function), 1308
nvs_sec_scheme_id_t (C++ enum), 1309
nvs_sec_scheme_id_t::NV$SEC_SCHEME_FLASH_ENC (C++ enumerator), 1309
nvs_sec_scheme_id_t::NV$SEC_SCHEME_HMAC (C++ enumerator), 1309
nvs_sec_scheme_id_t::NV$SEC_SCHEME_MAX (C++ enumerator), 1309
nvs_sec_scheme_t (C++ struct), 1289
nvs_sec_scheme_t::nv$flash_key_gen (C++ member), 1290
nvs_sec_scheme_t::nv$flash_read_cfg (C++ member), 1290
nvs_sec_scheme_t::scheme_data (C++ member), 1290
nvs_sec_scheme_t::scheme_id (C++ member), 1289
nvs_set_blob (C++ function), 1295
nvs_set_i16 (C++ function), 1291
nvs_set_i32 (C++ function), 1291
nvs_set_i64 (C++ function), 1291
nvs_set_i8 (C++ function), 1290
nvs_set_str (C++ function), 1291
nvs_set_u16 (C++ function), 1291
nvs_set_u32 (C++ function), 1291
nvs_set_u64 (C++ function), 1291
nvs_set_u8 (C++ function), 1291
nvs_stats_t (C++ struct), 1299
nvs_stats_t::available_entries (C++ member), 1300
nvs_stats_t::free_entries (C++ member), 1300
nvs_stats_t::namespace_count (C++ member), 1300
nvs_stats_t::total_entries (C++ member), 1300
nvs_stats_t::used_entries (C++ member), 1300
nvs_type_t (C++ enum), 1302
nvs_type_t::NV$TYPE_ANY (C++ enumerator), 1303
nvs_type_t::NV$TYPE_BLOB (C++ enumerator), 1303
nvs_type_t::NV$TYPE_I16 (C++ enumerator), 1303
nvs_type_t::NV$TYPE_I32 (C++ enumerator), 1303
nvs_type_t::NV$TYPE_I64 (C++ enumerator), 1303
nvs_type_t::NV$TYPE_I8 (C++ enumerator), 1302
nvs_type_t::NVS_TYPE_STR (C++ enumerator), 1303
nvs_type_t::NVS_TYPE_U16 (C++ enumerator), 1303
nvs_type_t::NVS_TYPE_U32 (C++ enumerator), 1303
nvs_type_t::NVS_TYPE_U64 (C++ enumerator), 1303
nvs_type_t::NVS_TYPE_U8 (C++ enumerator), 1302

pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_INCREASE (C++ enumerator), 640
pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_HALT (C++ enumerator), 640
pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_HOLD (C++ enumerator), 640
pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_KEEP (C++ enumerator), 640
pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_INVERSE (C++ enumerator), 640
pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_REVERSE (C++ enumerator), 640
pcnt_channel_level_action_t::PCNT_CHANNELLEVEL_ACTION_REVERT (C++ enumerator), 640
pcnt_channel_set_edge_action (C++ function), 637
pcnt_channel_set_level_action (C++ function), 637
pcnt_del_channel (C++ function), 637
pcnt_del_unit (C++ function), 632
pcnt_event_callbacks_t (C++ struct), 638
pcnt_glitch_filter_config_t (C++ struct), 639
pcnt_glitch_filter_config_t::max_glitch_ns (C++ member), 639
pcnt_new_channel (C++ function), 637
pcnt_new_unit (C++ function), 632
pcnt_unit_add_watch_point (C++ function), 636
pcnt_unit_clear_count (C++ function), 635
pcnt_unit_config_t::accum_count (C++ member), 639
pcnt_unit_config_t::flags (C++ member), 639
pcnt_unit_config_t::high_limit (C++ member), 638
pcnt_unit_config_t::intr_priority (C++ member), 638
pcnt_unit_config_t::low_limit (C++ member), 638
pcnt_unit_disable (C++ function), 634
pcnt_unit_enable (C++ function), 633
pcnt_unit_get_count (C++ function), 635
pcnt_unit_handle_t (C++ type), 640
pcnt_unit_register_event_callbacks (C++ function), 636
pcnt_unit_remove_watch_point (C++ function), 636
pcnt_unit_set_glitch_filter (C++ function), 633
pcnt_unit_update_glitch_filter (C++ function), 633
pcnt_unit_update_glitch_filter (C++ function), 633
pcnt_unit_update_glitch_filter (C++ function), 633
pcnt_unit_update_glitch_filter (C++ function), 633
pcnt_unit_update_glitch_filter (C++ function), 633
pcnt_watch_cb_t (C++ type), 640
pcnt_watch_event_data_t (C++ struct), 638
pcnt_watch_event_data_t::watch_point_value 
(C++ member), 638
pcnt_watch_event_data_t::zero_cross_mode 
(C++ member), 638
pcQueueGetName (C++ function), 1483
pcTaskGetName (C++ function), 1458
pcTTimerGetName (C++ function), 1517
PendedFunction_t (C++ type), 1527
phy_802_3_t (C++ struct), 389
phy_802_3_t::addr (C++ member), 389
phy_802_3_t::autonego_timeout_ms 
(C++ member), 389
phy_802_3_t::eth (C++ member), 389
phy_802_3_t::link_status (C++ member), 389
phy_802_3_t::parent (C++ member), 389
phy_802_3_t::reset_gpio_num (C++ member), 389
phy_802_3_t::reset_timeout_ms 
(C++ member), 389
phy_init_param_set (C++ function), 1991
phy_wifi_enable_set (C++ function), 1991
PIN_LEN (C macro), 344
print_class_descriptor_cb (C++ type), 885
protocomm_add_endpoint (C++ function), 1228
protocomm_ble_config (C++ struct), 1238
protocomm_ble_config::ble_bonding 
(C++ member), 1238
protocomm_ble_config::ble_link_encryption 
(C++ member), 1239
protocomm_ble_config::ble_sm_sc 
(C++ member), 1238
protocomm_ble_config::device_name 
(C++ member), 1238
protocomm_ble_config::manufacturer_data 
protocomm_security1_params::data 
(C++ member), 1232
protocomm_ble_config::manufacturer_data 
protocomm_security1_params::len 
(C++ member), 1232
protocomm_ble_config::nu_lookup 
(C++ member), 1238
protocomm_ble_config::nu_lookup_count 
(C++ member), 1238
protocomm_ble_config::service_uuid 
(C++ member), 1238
protocomm_ble_config::t (C++ type), 1239
protocomm_ble_event_t (C++ struct), 1238
protocomm_ble_event_t::conn_handle 
(C++ member), 1238
protocomm_ble_event_t::conn_status 
(C++ member), 1238
protocomm_ble_event_t::disconnect_reason 
protocomm_security2_params_t (C++ type), 1238
protocomm_ble_event_t::evt_type 
(C++ member), 1238
protocomm_ble_name_uuid_t (C++ type), 1239
protocomm_ble_start (C++ function), 1237
protocomm_ble_stop (C++ function), 1237
protocomm_close_session (C++ function), 1229
protocomm_delete (C++ function), 1228
protocomm_http_server_config_t::port 
(C++ member), 1236
protocomm_http_server_config_t::stack_size 
(C++ member), 1236
protocomm_http_server_config_t::task_priority 
(C++ member), 1236
protocomm_httpd_config_data_t 
(C++ union), 1236
protocomm_httpd_config_data_t::config 
(C++ member), 1236
protocomm_httpd_config_data_t::handle 
(C++ member), 1236
protocomm_httpd_config_data_t::data 
(C++ member), 1236
protocomm_httpd_config_t::ext_handle_provided 
(C++ member), 1236
PROTOCOMM_HTTPD_DEFAULT_CONFIG 
(C macro), 1236
protocomm_httpd_start (C++ function), 1235
protocomm_httpd_stop (C++ function), 1235
protocomm_new (C++ function), 1228
protocomm_protocols::open_session 
(C++ function), 1229
protocomm_remove_endpoint (C++ function), 1229
protocomm_req_handler (C++ function), 1230
protocomm_req_handler_t (C++ type), 1232
protocomm_security (C++ struct), 1233
protocomm_security1_params::data 
(C++ member), 1232
protocomm_security1_params::len 
(C++ member), 1232
protocomm_security2_params::data 
(C++ member), 1232
protocomm_security2_params::salt 
(C++ member), 1232
protocomm_security2_params::salt_len 
(C++ member), 1232
protocomm_security2_params::verifier 
(C++ member), 1233
protocomm_security2_params::verifier_len 
(C++ member), 1233
protocomm_security::cleanup 
(C++ member), 1233
protocomm_security::close_transport_session 
(C++ member), 1233
protocomm_security::decrypt 
(C++ mem-
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmt_new_rx_channel</td>
<td>C++ function</td>
<td>659</td>
</tr>
<tr>
<td>rmt_new_sync_manager</td>
<td>C++ function</td>
<td>656</td>
</tr>
<tr>
<td>rmt_new_tx_channel</td>
<td>C++ function</td>
<td>655</td>
</tr>
<tr>
<td>rmt_receive</td>
<td>C++ function</td>
<td>659</td>
</tr>
<tr>
<td>rmt_receive_config_t</td>
<td>C++ struct</td>
<td>661</td>
</tr>
<tr>
<td>rmt_receive_config_t::signal_range_max_ns</td>
<td>(C++ member)</td>
<td>662</td>
</tr>
<tr>
<td>rmt_receive_config_t::signal_range_min_ns</td>
<td>(C++ member)</td>
<td>662</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t</td>
<td>C++ struct</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::clk_src</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::flags</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::gpio_num</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::intr_priority</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::invert_in</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::io_loop_back</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::mem_block_symbols</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::resolution_hz</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_channel_config_t::with_dma</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_done_callback_t</td>
<td>(C++ type)</td>
<td>667</td>
</tr>
<tr>
<td>rmt_rx_done_event_data_t</td>
<td>(C++ struct)</td>
<td>666</td>
</tr>
<tr>
<td>rmt_rx_done_event_data_t::num_symbols</td>
<td>(C++ member)</td>
<td>667</td>
</tr>
<tr>
<td>rmt_rx_done_event_data_t::received_symbols</td>
<td>(C++ member)</td>
<td>667</td>
</tr>
<tr>
<td>rmt_rx_event_callbacks_t</td>
<td>(C++ struct)</td>
<td>666</td>
</tr>
<tr>
<td>rmt_rx_event_callbacks_t::on_recv_done</td>
<td>(C++ member)</td>
<td>661</td>
</tr>
<tr>
<td>rmt_rx_register_event_callbacks</td>
<td>(C++ function)</td>
<td>660</td>
</tr>
<tr>
<td>rmt_symbol_word_t</td>
<td>(C++ union)</td>
<td>667</td>
</tr>
<tr>
<td>rmt_symbol_word_t::duration0</td>
<td>(C++ member)</td>
<td>668</td>
</tr>
<tr>
<td>rmt_symbol_word_t::duration1</td>
<td>(C++ member)</td>
<td>668</td>
</tr>
<tr>
<td>rmt_symbol_word_t::level0</td>
<td>(C++ member)</td>
<td>668</td>
</tr>
<tr>
<td>rmt_symbol_word_t::level1</td>
<td>(C++ member)</td>
<td>668</td>
</tr>
<tr>
<td>rmt_symbol_word_t::val</td>
<td>(C++ member)</td>
<td>668</td>
</tr>
<tr>
<td>rmt_symbol_word_t::[anonymous]</td>
<td>(C++ member)</td>
<td>668</td>
</tr>
<tr>
<td>rmt_sync_manager_config_t</td>
<td>(C++ struct)</td>
<td>659</td>
</tr>
<tr>
<td>rmt_sync_manager_config_t::array_size</td>
<td>(C++ member)</td>
<td>659</td>
</tr>
<tr>
<td>rmt_sync_manager_config_t::tx_channel</td>
<td>(C++ member)</td>
<td>659</td>
</tr>
<tr>
<td>rmt_sync_manager_config_t::tx_channel_config</td>
<td>(C++ member)</td>
<td>659</td>
</tr>
<tr>
<td>rmt_sync_reset</td>
<td>(C++ function)</td>
<td>657</td>
</tr>
<tr>
<td>rmt_transmit</td>
<td>(C++ function)</td>
<td>655</td>
</tr>
<tr>
<td>rmt_transmit_config_t</td>
<td>(C++ struct)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_transmit_config_t::eot_level</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_transmit_config_t::flags</td>
<td>(C++ member)</td>
<td>659</td>
</tr>
<tr>
<td>rmt_transmit_config_t::loop_count</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_transfer_config_t::queue_nonblocking</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t</td>
<td>(C++ struct)</td>
<td>657</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::clk_src</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::flags</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::gpio_num</td>
<td>(C++ member)</td>
<td>657</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::intr_priority</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::invert_out</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::io_loop_back</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::io_od_mode</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::mem_block_symbols</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_channel_config_t::resolution_hz</td>
<td>(C++ member)</td>
<td>658</td>
</tr>
<tr>
<td>rmt_tx_done_callback_t</td>
<td>(C++ type)</td>
<td>657</td>
</tr>
<tr>
<td>rmt_tx_done_event_data_t</td>
<td>(C++ struct)</td>
<td>666</td>
</tr>
<tr>
<td>rmt_tx_done_event_data_t::num_symbols</td>
<td>(C++ member)</td>
<td>666</td>
</tr>
<tr>
<td>rmt_tx_event_callbacks_t</td>
<td>(C++ struct)</td>
<td>657</td>
</tr>
<tr>
<td>rmt_tx_event_callbacks_t::on_recv_done</td>
<td>(C++ member)</td>
<td>657</td>
</tr>
<tr>
<td>rmt_tx_register_event_callbacks</td>
<td>(C++ function)</td>
<td>656</td>
</tr>
<tr>
<td>rmt_tx_wait_all_done</td>
<td>(C++ function)</td>
<td>656</td>
</tr>
<tr>
<td>rtc_gpio_deinit</td>
<td>(C++ function)</td>
<td>504</td>
</tr>
<tr>
<td>rtc_gpio_force_hold_dis_all</td>
<td>(C++ function)</td>
<td>506</td>
</tr>
<tr>
<td>rtc_gpio_force_hold_en_all</td>
<td>(C++ function)</td>
<td>506</td>
</tr>
<tr>
<td>rtc_gpio_get_drive_capability</td>
<td>(C++ function)</td>
<td>507</td>
</tr>
<tr>
<td>rtc_gpio_get_level</td>
<td>(C++ function)</td>
<td>504</td>
</tr>
<tr>
<td>rtc_gpio_hold_dis</td>
<td>(C++ function)</td>
<td>506</td>
</tr>
<tr>
<td>rtc_gpio_hold_en</td>
<td>(C++ function)</td>
<td>506</td>
</tr>
<tr>
<td>rtc_gpio_init</td>
<td>(C++ function)</td>
<td>503</td>
</tr>
<tr>
<td>rtc_gpio_iOMUX_func_sel</td>
<td>(C++ function)</td>
<td>505</td>
</tr>
<tr>
<td>RTC_GPIO-IS-VALID_GPIO</td>
<td>(C macro)</td>
<td>507</td>
</tr>
<tr>
<td>rtc_gpio_is_valid_gpio</td>
<td>(C++ function)</td>
<td>503</td>
</tr>
<tr>
<td>rtc_gpio_mode_t</td>
<td>(C++ enum)</td>
<td>507</td>
</tr>
</tbody>
</table>
rtc_gpio_mode_t::RTC_GPIO_MODE_DISABLE
sdmmc_io_print_cis_info (C++ function),
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_INPUT_ON
sdmmc_io_read_blocks (C++ function), 1320
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_INPUT_OFF
sdmmc_io_read_bytes (C++ function), 1320
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT
sdmmc_remove_device (C++ function), 1320
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_IN
sdmmc_open_blocks (C++ function), 1320
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_OUT
sdmmc_mmc_can_sanitize (C++ function), 1319
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_IN_OUT
sdmmc_mmcSanitize (C++ function), 1319
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_DISABLE
sdmmc_read_sectors (C++ function), 1318
(C++ enumerator), 507
rtc_gpio_mode_t::RTC_GPIO_MODE_OUTPUT_DISABLE_IN
SDSPI_DEFAULT_DMA (C macro), 673
tsc_gpio_write_disable (C++ function), 1318
(C++ function), 674
SDSPI_DEFAULT_HOST (C macro), 673
tsc_gpio_write_enable (C++ function), 1318
tsc_gpio_dev_handle_t (C++ type), 674
SDSPI_DEVICE_CONFIG_DEFAULT (C macro), 674
tsc_gpio_dev_config_t (C++ struct), 673
tsc_gpio_dev_config_t::gpio_cd (C++ member), 673
tsc_gpio_dev_config_t::gpio_cs (C++ member), 673
tsc_gpio_dev_config_t::gpio_int (C++ member), 673
tsc_gpio_dev_config_t::gpio_wp (C++ member), 673
tsc_gpio_dev_config_t::gpio_wp_polarity (C++ member), 673
tsc_gpio_dev_config_t::host_id (C++ member), 673
SDSPI_HOST_DEFAULT (C macro), 673
tsc_gpio_host_deinit (C++ function), 672
SDSPI_HOST_DO_TRANSACTION (C++ function), 671
tsc_gpio_host_get_real_freq (C++ function), 672
tsc_gpio_host_init (C++ function), 671
tsc_gpio_host_init_device (C++ function), 671
tsc_gpio_host_io_int_enable (C++ function), 672
tsc_gpio_host_io_int_wait (C++ function), 672
tsc_gpio_host_remove_device (C++ function), 671
SDSPIHOST_SET_CARD_CLK (C++ function), 672
SDSPI_IO_ACTIVE_LOW (C macro), 674
SDSPI_SLOT_NO_CD (C macro), 673
SDSPI_SLOT_NO_CS (C macro), 673
SDSPI_SLOT_NO_INT (C macro), 673
SemaphoreHandle_t (C++ type), 1510
semBINARY_SEMAPHORE_QUEUE_LENGTH (C macro), 1496
semGIVE_BLOCK_TIME (C macro), 1496
semSEMAPHORE_QUEUE_ITEM_LENGTH (C macro), 1496
shared_stack_function (C++ type), 1370
shutdown_handler_t (C++ type), 1648
Index

slave_cb_t (C++ type), 749
slave_transactions_cb_t (C++ type), 741
smartconfig_event_on_get_ssid_pswd_t
(C++ struct), 269
smartconfig_event_on_get_ssid_pswd_t::bssid
(C++ member), 269
smartconfig_event_on_get_ssid_pswd_t::bssid_set
(C++ member), 269
smartconfig_event_on_get_ssid_pswd_t::cellphone
(C++ member), 270
smartconfig_event_on_get_ssid_pswd_t::password
(C++ member), 270
smartconfig_event_on_get_ssid_pswd_t::token
(C++ member), 270
smartconfig_event_on_get_ssid_pswd_t::type
(C++ member), 270
smartconfig_event_on_get_ssid_pswd_t::token
(C++ member), 270
smartconfig_event_on_get_ssid_pswd_t::type
(C++ member), 270
smartconfig_event_t (C++ enumber), 270
smartconfig_event_t::SC_EVENT_FOUND_CHANNEL
(C++ enumber), 271
smartconfig_event_t::SC_EVENT_GET_SSID_PSWD
(C++ enumber), 271
smartconfig_event_t::SC_EVENT_SCAN_DONE
(C++ enumber), 270
smartconfig_event_t::SC_EVENT_SEND_ACK_DONE
(C++ enumber), 271
SMARTCONFIG_START_CONFIG_DEFAULT
(C macro), 270
smartconfig_start_config_t
(C++ struct), 270
smartconfig_start_config_t::enable_log
(C++ member), 270
smartconfig_start_config_t::esp_touch_v2_enable
(C++ member), 270
smartconfig_start_config_t::esp_touch_v2_enable
(C++ member), 270
smartconfig_type_t (C++ enumber), 270
smartconfig_type_t::SC_TYPE_AIRKISS
(C++ enumber), 270
smartconfig_type_t::SC_TYPE_ESPTOUCH
(C++ enumber), 270
smartconfig_type_t::SC_TYPE_ESPTOUCH_AUTH
(C++ enumber), 270
smartconfig_type_t::SC_TYPE_ESPTOUCH_V2
(C++ enumber), 270
snntp_set_time_sync_notification_cb
(C++ function), 1723
snntp_setoperatingmode (C++ function), 1724
snntp_setservername (C++ function), 1724
snntp_servermode (C++ function), 1725
snntp_syncmode (C++ function), 1725
snntp_syncmode_t::SNTP_SYNC_MODE_IMMED
(C++ function), 1725
snntp_syncmode_t::SNTP_SYNC_MODE_SMOOTH
(C++ function), 1725
snntp_sync_status_t::SNTP_SYNC_STATUS_COMPLETED
(C++ function), 1725
snntp_sync_status_t::SNTP_SYNC_STATUS_IN_PROGRESS
(C++ function), 1725
snntp_sync_status_t::SNTP_SYNC_STATUS_RESET
(C++ function), 1725
SOC_ADC_ABORT_SUPPORTED (C macro), 1708
SOC_ADC_ATTEN_NUM (C macro), 1708
SOC_ADC_CALIBRATION_V1_SUPPORTED (C
macro), 1709
SOC_ADC_CHANNEL_NUM (C macro), 1708
SOC_ADC_CLKS (C macro), 465
SOC_ADC_CONTROLLER_NUM (C macro), 1708
SOC_ADC_DATA_BYTES_PER_CONV (C
macro), 1709
SOC_ADC_DIGI_FILTER_NUM (C macro), 1708
SOC_ADC_DIGI_MAX_BIT_WIDTH (C macro), 1708
SOC_ADC_DIGI_MIN_BIT_WIDTH (C macro), 1708
SOC_ADC_DIGI_MONITOR_NUM (C macro), 1709
SOC_ADC_DIGI_RESULT_BYTES (C macro), 1709
SOC_ADC_DMA_SUPPORTED (C macro), 1708
SOC_ADC_MAX_CHANNEL_NUM (C macro), 1708
SOC_ADC_MAX_FREQ_THRES_HIGH (C
macro), 1709
SOC_ADC_MAX_FREQ_THRES_LOW (C macro), 1709
SOC_ADC_PERIPH_NUM (C macro), 1708
SOC_ADC_RTC_CLKS (C macro), 465
SOC_ADC_RTC_CTRL_SUPPORTED (C macro), 1708
SOC_ADC_RTC_MAX_BIT_WIDTH (C macro), 1709
SOC_ADC_RTC_MIN_BIT_WIDTH (C macro), 1709
SOC_ADC_SAMPLE_FREQ_THRES_HIGH (C
macro), 1709
SOC_ADC_SAMPLE_FREQ_THRES_LOW (C macro), 1709
SOC_ADC_SELF_HW_CALI_SUPPORTED (C
macro), 1709
SOC_ADC_SHARED_POWER (C macro), 1709
SOC_ADC_SUPPORTED (C macro), 1706
SOC_AES_CRYPTO_DMA (C macro), 1717
SOC_AES_SUPPORT_AES_128 (C macro), 1717
SOC_AES_SUPPORT_AES_192 (C macro), 1717
SOC_AES_SUPPORT_AES_256 (C macro), 1717
SOC_AES_SUPPORT_DMA (C macro), 1716
SOC_AES_SUPPORT_GCM (C macro), 1716
SOC_AES_SUPPORTED (C macro), 1707
SOC_ASYNC_MEMCPY_SUPPORTED (C macro), 1706
SOC_BOD_SUPPORTED (C macro), 1707
SOC_BROWNOUT_RESET_SUPPORTED (C macro), 1709
SOC_CACHE_SUPPORT_WRAP (C macro), 1706
SOC_CACHE_WRITEBACK_SUPPORTED (C macro), 1709
SOC_CCOMP_TIMER_SUPPORTED (C macro), 1706
SOC_CLK_XTAL32K_SUPPORTED (C macro), 1708
SOC_CLK_TREE_SUPPORTED (C macro), 1706
SOC_CLK_RC_FAST_FREQ_APPROX (C macro), 464
SOC_CLK_RC_FAST_D256_FREQ_APPROX (C macro), 464
SOC_CLK_RC_FAST_D256_SUPPORTED (C macro), 1718
SOC_CLK_RC_FAST_FREQ_APPROX (C macro), 464
SOC_CLK_RC_FAST_SUPPORT_CALIBRATION (C macro), 1718
SOC_CLK_RC_SLOW_FREQ_APPROX (C macro), 464
SOC_CLK_TREE_SUPPORTED (C macro), 1708
SOC_CLK_XTAL32K_FREQ_APPROX (C macro), 464
SOC_CLK_XTAL32K_SUPPORTED (C macro), 1718
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_APLL (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_INVALID (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_PLL (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_PLL_FAST (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_RC_FAST (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_RC_SLOW (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_REF_TICK (C++ enumerator), 473
SOC_CLKOUT_SIG_ID_t::CLKOUT_SIG_XTAL (C++ enumerator), 473
SOC_COEX_HW_PTI (C macro), 1718
SOC_CONFIGURABLE_VDDSDIO_SUPPORTED (C macro), 1718
SOC_CP_DMA_MAX_BUFFER_SIZE (C macro), 1709
SOC_CP_DMA_SUPPORTED (C macro), 1706
SOC_CPU_BREAKPOINTS_NUM (C macro), 1709
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_APLL (C++ enumerator), 466
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_APISOCFLASH_ENCRYPTION_XTS_AES_128 (C macro), 1717
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_INVALID (C++ enumerator), 466
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_INVFLASH_ENCRYPTION_XTS_AES_256 (C macro), 1717
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_PLL (C++ enumerator), 466
soc_cpu_clk_src_t::SOC_CPU_CLK_SRC_XTAL (C++ enumerator), 466
SOC_CPU_CORES_NUM (C macro), 1709
SOC_CPU_INTR_NUM (C macro), 1709
SOC_CPU_WATCHPOINT_MAX_REGION_SIZE (C macro), 1709
SOC_CPU_WATCHPOINTS_NUM (C macro), 1709
SOC_DAC_CHAN_NUM (C macro), 1709
SOC_DAC_COSINE_CLKS (C macro), 465
SOC_DAC_DIGI_CLKS (C macro), 465
SOC_DAC_RESOLUTION (C macro), 1709
SOC_DAC_SUPPORTED (C macro), 1706
SOC_DIGI_SIGN_SUPPORTED (C macro), 1706
SOC_DEDIC_GPIO_ALLOW_REG_ACCESS (C macro), 1710
SOC_DEDIC_GPIO_HAS_INTERRUPT (C macro), 1710
SOC_DEDIC_GPIO_IN_CHANNELS_NUM (C macro), 1710
SOC_DEDIC_GPIO_OUT_AUTO_ENABLE (C macro), 1710
SOC_DEDIC_GPIO_OUT_CHANNELS_NUM (C macro), 1710
SOC_DEDGED_GPIO_SUPPORTED (C macro), 1706
SOC_DIGI_SIGN_SUPPORTED (C macro), 1707
SOC_EFUSE_DIS_JTAG (C macro), 1716
SOC_EFUSE_DIS_LEGACY_SPI_BOOT (C macro), 1717
SOC_EFUSE_DIS_DOWNLOAD_ICACHE (C macro), 1716
SOC_EFUSE_DIS_DOWNLOAD_DCACHE (C macro), 1716
SOC_EFUSE_DIS_DOWNLOAD_DCACHE (C macro), 1716
SOC_EFUSE_DIS_ICACHE (C macro), 1717
SOC_EFUSE_DIS_ICACHE (C macro), 1717
SOC_EFUSE_DIS_LEGACY_SPI_BOOT (C macro), 1717
SOC_EFUSE_DISK_SIGN_SUPPORTED (C macro), 1706
SOC_EFUSE_HARD_DIS_JTAG (C macro), 1716
SOC_EFUSE_KEY_PURPOSE_FIELD (C macro), 1706
SOC_EFUSE_SECURE_BOOT_KEY_DIGESTS (C macro), 1717
SOC_EFUSE_SECURE_BOOT_KEY_DIGESTS (C macro), 1717
SOC_EXTERNAL_COEX_ADVANCE (C macro), 1718
SOC_EXTERNAL_COEX_LEADER_TX_LINE (C macro), 1718
SOC_FLASH_ENC_SUPPORTED (C macro), 1707
SOC_FLASH_ENCRYPTED_XTS_AES_BLOCK_MAX (C macro), 1717
SOC_FLASH_ENCRYPTION_XTS_AES (C macro), 1717
SOC_FLASH_ENCRYPTION_XTS_AES_128 (C macro), 1717
SOC_FLASH_ENCRYPTION_XTS_AES_256 (C macro), 1717
SOC_FLASH_ENCRYPTION_XTS_AES_OPTIONS (C macro), 1717
SOC_FLASH_ENCRYPTION_XTS_AES_OPTIONS (C macro), 1717
Index

SOC_GLITCH_FILTER_CLKS (C macro), 1717
SOC_GPIO_CLOCKOUT_BY_IO_MUX (C macro), 1710
SOC_GPIO_FILTER_CLK_SUPPORT_APB (C macro), 1710
SOC_GPIO_IN_RANGE_MAX (C macro), 1710
SOC_GPIO_OUT_RANGE_MAX (C macro), 1710
SOC_GPIO_PIN_COUNT (C macro), 1710
SOC_GPIO_PORT (C macro), 1710
SOC_GPIO_SUPPORT_FORCE_HOLD (C macro), 1710
SOC_GPIO_SUPPORT_PIN_GLITCH_FILTER (C macro), 1710
SOC_GPIO_SUPPORT_RTC_INDEPENDENT (C macro), 1710
SOC_GPIO_VALID_DIGITAL_IO_PAD_MASK (C macro), 1710
SOC_GPIO_VALID_GPIO_MASK (C macro), 1710
SOC_GPIO_SUPPORT_PIN_GLITCH_FILTER (C macro), 1710
SOC_GPIO_VALID_OUTPUT_GPIO_MASK (C macro), 1710
SOC_MPU_REGIONS_MAX_NUM (C macro), 1710
SOC_MPU_REGION_WO_SUPPORTED (C macro), 1710
SOC_MPU_REGION_RO_SUPPORTED (C macro), 1710
SOC_MPU_MEM_BLOCKS_NUM (C macro), 1710
soc_module_clk_t::SOC_MOD_CLK_XTAL32K (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_APLL (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_INVALID (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_CPU (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_RC_FAST (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_RC_FAST_D256 (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_REF_TICK (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_RTC_FAST (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_RTC_SLOW (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_TEMP_SENSOR (C++ enumerator), 468
soc_module_clk_t::SOC_MOD_CLK_XTAL (C++ enumerator), 467
soc_module_clk_t::SOC_MOD_CLK_XTAL32K (C++ enumerator), 467
SOC_MPI_MEM_BLOCKS_NUM (C macro), 1716
SOC_MPI_OPERATIONS_NUM (C macro), 1716
SOC_MPI_SUPPORTED (C macro), 1707
SOC_MPU_MEMORY_REGIONS_SUPPORTED (C macro), 1712
SOC_MPU_MIN_REGION_SIZE (C macro), 1712
SOC_MPU_REGIONleston_SUPPORTED (C macro), 1712
SOC_MPU_REGION_WO_SUPPORTED (C macro), 1712
SOC_MPU_REGIONS_MAX_NUM (C macro), 1712

Espressif Systems

Submit Document Feedback

Release master
Index

SOC_MPU_SUPPORTED (C macro), 1708
SOC_MWDT_CLKS (C macro), 465
SOC_PCIE_CHANNELS_PER_UNIT (C macro), 1712
SOC_PCIE_GROUPS (C macro), 1712
SOC_PCIE_SUPPORTED (C macro), 1706
SOC_PCIE_THRES_POINT_PER_UNIT (C macro), 1712
SOC_PCIE_UNITS_PER_GROUP (C macro), 1712

soc_periph_i2s_clk_src_t (C++ enumerator), 470
soc_periph_i2s_clk_src_t::I2S_CLK_SRC_APPL (C++ enumerator), 470
soc_periph_i2s_clk_src_t::I2S_CLK_SRC_DEFAULT (C++ enumerator), 470
soc_periph_i2s_clk_src_t::I2S_CLK_SRC_PLL160M (C++ enumerator), 470
soc_periph_lcd_clk_src_t::LCD_CLK_SRC_DEFAULT (C++ enumerator), 469
soc_periph_lcd_clk_src_t::LCD_CLK_SRC_PLL160M (C++ enumerator), 469

soc_periph_adc_digi_clk_src_t::ADC_DIGI_CLK_SRC_APB (C++ enumerator), 472
soc_periph_adc_digi_clk_src_t::ADC_DIGI_CLK_SRC_DEFAULT (C++ enumerator), 472
soc_periph_adc_rtc_clk_src_t::ADC_RTC_CLK_SRC_DEFAULT (C++ enumerator), 472
soc_periph_adc_rtc_clk_src_t::ADC_RTC_CLK_SRC_PLL160M (C++ enumerator), 472

soc_periph_spi_clk_src_t::SPI_CLK_SRC_DEFAULT (C++ enumerator), 470
soc_periph_spi_clk_src_t::SPI_CLK_SRC_APB (C++ enumerator), 470

soc_periph_sdm_clk_src_t::SDM_CLK_SRC_DEFAULT (C++ enumerator), 470
soc_periph_sdm_clk_src_t::SDM_CLK_SRC_APB (C++ enumerator), 470

soc_periph_rmt_clk_src_t::RMT_CLK_SRC_REF_TICK (C++ enumerator), 472
soc_periph_rmt_clk_src_t::RMT_CLK_SRC_DEFAULT (C++ enumerator), 472
soc_periph_rmt_clk_src_t::RMT_CLK_SRC_APB (C++ enumerator), 472
soc_periph_rmt_clk_src_legacy_t::RMT_BASECLK_REF (C++ enumerator), 472
soc_periph_rmt_clk_src_legacy_t::RMT_BASECLK_APB (C++ enumerator), 472

soc_periph_systimer_clk_src_t::SYSTIMER_CLK_SRC_DEFAULT (C++ enumerator), 472

soc_periph_gptimer_clk_src_t::GPTIMER_CLK_SRC_DEFAULT (C++ enumerator), 469
soc_periph_gptimer_clk_src_t::GPTIMER_CLK_SRC_APB (C++ enumerator), 469

soc_periph_gptimer_clk_src_t::GPTIMER_CLK_SRC_DEFAULT (C++ enumerator), 469
soc_periph_gptimer_clk_src_t::GPTIMER_CLK_SRC_APB (C++ enumerator), 469

soc_periph_glitch_filter_clk_src_t::GLITCH_FILTER_CLK_SRC_DEFAULT (C++ enumerator), 469
soc_periph_glitch_filter_clk_src_t::GLITCH_FILTER_CLK_SRC_APB (C++ enumerator), 469

soc_periph_dac_digi_clk_src_t::DAC_DIGI_CLK_SRC_APB (C++ enumerator), 472
soc_periph_dac_digi_clk_src_t::DAC_DIGI_CLK_SRC_DEFAULT (C++ enumerator), 472

soc_periph_dac_cosine_clk_src_t::DAC_COSINE_CLK_SRC_RTC_FAST (C++ enumerator), 472
soc_periph_dac_cosine_clk_src_t::DAC_COSINE_CLK_SRC_DEFAULT (C++ enumerator), 472

soc_periph_lcd_clk_src_t::LCD_CLK_SRC_DEFAULT (C++ enumerator), 472
soc_periph_lcd_clk_src_t::LCD_CLK_SRC_PLL160M (C++ enumerator), 472

SOC_PCNT_UNITS_PER_GROUP
SOC_PCNT_THRES_POINT_PER_UNIT
SOC_PCNT_SUPPORTED
SOC_PCNT_GROUPS
SOC_PCNT_CHANNELS_PER_UNIT
SOC_MWDT_CLKS

Index
SOC_SECURE_BOOT_SUPPORTED (C macro), 1707
SOC_SECURE_BOOT_V2_RSA (C macro), 1717
SOC_SHA_CRYPTO_DMA (C macro), 1716
SOC_SHA_DMA_MAX_BUFFER_SIZE (C macro), 1716
SOC_SHA_SUPPORT_SECURE_DL_MODE (C macro), 1706
SOC_SHA_SUPPORT_SECURE_BOOT_REVOKE_KEY (C macro), 1717
SOC_SHA_DMA_MAX_BUFFER_SIZE (C macro), 1716
SOC_SHA_SUPPORT_DL_MODE (C macro), 1706
SOC_SYSTIMER_ALARM_NUM (C macro), 1714
SOC_SYSTIMER_BIT_WIDTH_HI (C macro), 1714
SOC_SYSTIMER_BIT_WIDTH_LO (C macro), 1714
SOC_SYSTIMER_COUNTER_NUM (C macro), 1714
SOC_SYSTIMER_SUPPORTED (C macro), 1707
SOC_TEMP_SENSOR_CLKS (C macro), 464
SOC_TEMP_SENSOR_SUPPORTED (C macro), 1706
SOC_TEMPERATURE_SENSOR_SUPPORT_FAST_RC (C macro), 1718
SOC_TIMER_GROUP_COUNTER_BIT_WIDTH (C macro), 1714
SOC_TIMER_GROUP_SUPPORT_APB (C macro), 1714
SOC_TIMER_GROUP_SUPPORT_XTAL (C macro), 1714
SOC_TIMER_GROUP_TIMERS_PER_GROUP (C macro), 1714
SOC_TIMER_GROUP_TOTAL_TIMERS (C macro), 1715
SOC_TIMER_GROUPS (C macro), 1714
SOC_TOUCH_PROXIMITY_CHANNEL_NUM (C macro), 1715
SOC_TOUCH_SAMPLER_NUM (C macro), 1715
SOC_TOUCH_SENSOR_NUM (C macro), 1715
SOC_TOUCH_SENSOR_SUPPORTED (C macro), 1707
SOC_TOUCH_SENSOR_VERSION (C macro), 1715
SOC_TWAI_BRP_MAX (C macro), 1715
SOC_TWAI_BRP_MIN (C macro), 1715
SOC_TWAI_CLK_SUPPORT_APB (C macro), 1715
SOC_TWAI_CLKS (C macro), 465
SOC_TWAI_CONTROLLER_NUM (C macro), 1715
SOC_TWAI_SUPPORTED (C macro), 1706
SOC_TWAI_SUPPORTS_RX_STATUS (C macro), 1715
SOC_UART_BDIRATE_MAX (C macro), 1715
SOC_UART_FIFO_LEN (C macro), 1715
SOC_UART_HP_NUM (C macro), 1715
SOC_UART_NUM (C macro), 1715
SOC_UART_SUPPORT_APB_CLK (C macro), 1715
SOC_UART_SUPPORT_REF_TICK (C macro), 1715
SOC_UART_SUPPORT_WAKEUP_INT (C macro), 1715
SOC_UART_SUPPORTED (C macro), 1706
SOC_ULP_FSM_SUPPORTED (C macro), 1706
SOC_ULP_HAS_ADC (C macro), 1719
SOC_ULP_SUPPORTED (C macro), 1706
SOC_USB_OTG_PERIPH_NUM (C macro), 1716
SOC_USB_OTG_SUPPORTED (C macro), 1706
SOC_WDT_SUPPORTED (C macro), 1708
SOC_WIFI_CSI_SUPPORT (C macro), 1719
SOC_WIFI_FTM_SUPPORT (C macro), 1719
SOC_WIFI_HW_TSF (C macro), 1719
SOC_WIFI_LIGHT_SLEEP_CLK_WIDTH (C macro), 1717
SOC_WIFI_MESH_SUPPORT (C macro), 1719
SOC_WIFI_NAN_SUPPORT (C macro), 1719
SOC_WIFI_SUPPORT_VARIABLE_BEACON_WINDOW (C macro), 1719
SOC_WIFI_SUPPORTED (C macro), 1706
SOC_WIFI_WAPI_SUPPORT (C macro), 1719
SPI_DEVICE_NO_DUMMY (C macro), 733
SPI_DEVICE_NO_RETURN_RESULT (C macro), 733
spi_device_polling_end (C++ function), 727
spi_device_polling_start (C++ function), 726
spi_device_polling_transmit (C++ function), 727
SPI_DEVICE_POSITIVE_CS (C macro), 732
spi_device_queue_trans (C++ function), 725
spi_device_release_bus (C++ function), 728
SPI_DEVICE_RXBIT_LSBFIRST (C macro), 732
spi_device_transmit (C++ function), 726
SPI_DEVICE_TXBIT_LSBFIRST (C macro), 732
spi_dma_chan_t (C++ type), 724
spi_event_t (C++ enum), 719
spi_event_t::SPI_EV_BUF_RX (C++ enumerator), 720
spi_event_t::SPI_EV_BUF_TX (C++ enumerator), 719
spi_event_t::SPI_EV_CMD9 (C++ enumerator), 720
spi_event_t::SPI_EV_CMDA (C++ enumerator), 720
spi_event_t::SPI_EV_RECV (C++ enumerator), 719
spi_event_t::SPI_EV_RECV_DMA_READY (C++ enumerator), 719
spi_event_t::SPI_EV_SEND (C++ enumerator), 719
spi_event_t::SPI_EV_SEND_DMA_READY (C++ enumerator), 719
spi_event_t::SPI_EV_TRANS (C++ enumerator), 720
spi_flash_cache2phys (C++ function), 699
SPI_FLASH_CACHE2PHYS_FAIL (C macro), 700
spi_flash_chip_t (C++ type), 698
SPI_FLASH_CONFIG_CONF_BITS (C macro), 705
spi_flash_counter_t (C++ type), 708
spi_flash_counters_t (C++ type), 708
spi_flash_dump_counters (C++ function), 707
spi_flash_encryption_t (C++ struct), 702
spi_flash_encryption_t::flash_encryption_enable (C++ member), 703
spi_flash_encryption_t::flash_encryption_done (C++ member), 703
spi_flash_encryption_t::flash_encryption_disable (C++ member), 703
spi_flash_encryption_t::flash_encryption_data_prepare (C++ member), 703
spi_flash_encryption_t::flash_encryption_check (C++ member), 703
spi_flash_host_driver_s::check_suspend (C++ member), 704
spi_flash_host_driver_s::common_command (C++ member), 703
spi_flash_host_driver_s::configure_host_io_mode (C++ member), 704
spi_flash_host_driver_s::dev_config (C++ member), 703
spi_flash_host_driver_s::erase_block (C++ member), 703
spi_flash_host_driver_s::erase_chip (C++ member), 703
spi_flash_host_driver_s::erase_sector (C++ member), 703
spi_flash_host_driver_s::flush_cache (C++ member), 704
spi_flash_host_driver_s::host_status (C++ member), 704
spi_flash_host_driver_s::poll_cmd_done (C++ member), 704
spi_flash_host_driver_s::program_page (C++ member), 703
spi_flash_host_driver_s::read (C++ member), 704
spi_flash_host_driver_s::read_data_slicer (C++ member), 704
spi_flash_host_driver_s::read_id (C++ member), 703
spi_flash_host_driver_s::read_status (C++ member), 703
spi_flash_host_driver_s::resume (C++ member), 704
spi_flash_host_driver_s::set_write_protect (C++ member), 703
spi_flash_host_driver_s::supports_direct_read (C++ member), 704
spi_flash_host_driver_s::supports_direct_write (C++ member), 703
spi_flash_host_driver_s::sus_setup (C++ member), 704
spi_flash_host_driver_s::suspend (C++ member), 704
spi_flash_host_driver_s::write_data_slicer (C++ member), 703
spi_flash_host_driver_t (C++ type), 705
spi_flash_host_inst_t (C++ struct), 702
spi_flash狉_host_mgmt (C++ function), 698
spi_flash_host_mgmt (C++ function), 699
spi_flash_host_mgmt (C++ function), 699
spi_flash_host_mgmt (C++ function), 699
spi_flash_host_mgmt (C++ function), 699
spi_flash_host_mgmt (C++ function), 699
spi_flash_mmap_pages (C++ function), 698
SPI_FLASH_MMU_PAGE_SIZE (C macro), 700

Espressif Systems 2291
Submit Document Feedback
Release master
spi_flash_munmap (C++ function), 699
SPI_FLASH_OPI_FLAG (C macro), 705
SPI_FLASH_OS_IS_ERASING_STATUS_FLAG (C macro), 698
spi_flash physically cache (C++ function), 699
SPI_FLASH_READ_MODE_MIN (C macro), 705
spi_flash_read_counter (C++ function), 707
SPI_FLASH_SEC_SIZE (C macro), 700
spi_flash_sus_cmd_conf (C++ struct), 701
spi_flash_sus_cmd_conf::cmd_rdsr (C++ member), 702
spi_flash_sus_cmd_conf::res_cmd (C++ member), 702
spi_flash_sus_cmd_conf::reserved (C++ member), 702
spi_flash_sus_cmd_conf::sus_cmd (C++ member), 702
spi_flash_sus_cmd_conf::sus_mask (C++ member), 701
SPI_FLASH_TRANS_FLAG_BYTE_SWAP (C macro), 705
SPI_FLASH_TRANS_FLAG_CMD16 (C macro), 704
SPI_FLASH_TRANS_FLAG_IGNORE_BASEIO (C macro), 704
SPI_FLASH_TRANS_FLAG_PE_CMD (C macro), 705
spi_flash_trans_t (C++ struct), 701
spi_flash_trans_t::address (C++ member), 701
spi_flash_trans_t::address_bitlen (C++ member), 701
spi_flash_trans_t::command (C++ member), 701
spi_flash_trans_t::dummy_bitlen (C++ member), 701
spi_flash_trans_t::flags (C++ member), 701
spi_flash_trans_t::io_mode (C++ member), 701
spi_flash_trans_t::miso_data (C++ member), 701
spi_flash_trans_t::miso_len (C++ member), 701
spi_flash_trans_t::mosi_data (C++ member), 701
spi_flash_trans_t::mosi_len (C++ member), 701
spi_flash_trans_t::reserved (C++ member), 701
SPI_FLASH_YIELD_REQ_SUSPEND (C macro), 698
SPI_FLASH_YIELD_REQ_YIELD (C macro), 698
SPI_FLASH_YIELD_STA_RESUME (C macro), 698
spi_get_actual_clock (C++ function), 728
spi_get_freq_limit (C++ function), 728
spi_get_timing (C++ function), 728
spi_host_device_t::SPI_HOST (C++ enumerator), 719
spi_host_device_t::SPI1_HOST (C++ enumerator), 719
spi_host_device_t::SPI2_HOST (C++ enumerator), 719
spi_host_device_t::SPI3_HOST (C++ enumerator), 719
spi_host_device_t::SPI_HOST_MAX (C++ enumerator), 719
spi_line_mode_t (C++ struct), 718
spi_line_mode_t::addr_lines (C++ member), 719
spi_line_mode_t::cmd_lines (C++ member), 718
spi_line_mode_t::data_lines (C++ member), 719
SPI_MASTER_FREQ_10M (C macro), 732
SPI_MASTER_FREQ_11M (C macro), 732
SPI_MASTER_FREQ_13M (C macro), 732
SPI_MASTER_FREQ_16M (C macro), 732
SPI_MASTER_FREQ_20M (C macro), 732
SPI_MASTER_FREQ_26M (C macro), 732
SPI_MASTER_FREQ_40M (C macro), 732
SPI_MASTER_FREQ_80M (C macro), 732
SPI_MASTER_FREQ_90M (C macro), 732
SPI_MAX_DMA_LEN (C macro), 723
SPI_SLAVE_BIT_LSBFIRST (C macro), 740
spi_slave_chan_t (C++ enum), 749
spi_slave_chan_t::SPI_SLAVE_CHAN_RX (C++ enumerator), 749
spi_slave_chan_t::SPI_SLAVE_CHAN_TX (C++ enumerator), 749
spi_slave_free (C++ function), 738
spi_slave_get_trans_result (C++ function), 738
SPI_SLAVE HD_APPEND_MODE (C macro), 748
spi_slave_hd_append_trans (C++ function), 745
SPI_SLAVE HD㸌 לל (C macro), 748
spi_slave_hd_callback_config_t (C++ struct), 747
spi_slave_hd_callback_config_t::cb_arg (C++ member), 747
spi_slave_hd_callback_config_t::cb_buffer_rx (C++ member), 747
spi_slave_hd_callback_config_t::cb_buffer_tx (C++ member), 747
spi_slave_hd_callback_config_t::cb_cmd9 (C++ member), 747
spi_slave_hd_callback_config_t::cb_cmdA (C++ member), 747
spi_slave_hd_callback_config_t::cb_recv (C++ member), 747
spi_slave_hd_callback_config_t::cb_recv_dma_ready (C++ member), 747
spi_slave_hd_callback_config_t::cb_send_dma_ready (C++ member), 747
spi_slave_hd_callback_config_t::cb_sent

Espressif Systems 2292 Release master
Submit Document Feedback
spi_slave_interface_config_t::flags
(C++ member), 739
spi_slave_interface_config_t::mode
(C++ member), 739
spi_slave_interface_config_t::post_setup_cb
(C++ member), 740
spi_slave_interface_config_t::post_trans_cb
(C++ member), 740
spi_slave_interface_config_t::queue_size
(C++ member), 739
spi_slave_interface_config_t::spics_io_num
(C++ member), 739
SPI_SLAVE_NO_RETURN_RESULT (C macro), 740
spi_slave_queue_trans (C++ function), 738
SPI_SLAVE_RXBIT_LSBFIRST (C macro), 740
SPI_SLAVE_TRANS_DMA_BUFFER_ALIGN_AUTO
(C macro), 741
spi_slave_transaction_t::trans (C++ struct), 740
spi_slave_transaction_t::flags (C++ member), 740
spi_slave_transaction_t::length (C++ member), 740
spi_slave_transaction_t::rx_buffer
(C++ member), 740
spi_slave_transaction_t::trans_len
(C++ member), 740
spi_slave_transaction_t::tx_buffer
(C++ member), 740
spi_slave_transaction_t::user (C++ member), 740
spi_slave_transmit (C++ function), 739
SPI_SLAVE_TXBIT_LSBFIRST (C macro), 740
SPI_SWAP_DATA_RX (C macro), 723
SPI_SWAP_DATA_TX (C macro), 723
SPI_TRANS_CS_KEEP_ACTIVE (C macro), 733
SPI_TRANS_DMA_BUFFER_ALIGN_MANUAL (C macro), 734
SPI_TRANS_MODE_DIO (C macro), 733
SPI_TRANS_MODE_DIOQIO_ADDR (C macro), 733
SPI_TRANS_MODE_OCT (C macro), 734
SPI_TRANS_MODE_QIO (C macro), 733
SPI_TRANS_MULTILINE_ADDR (C macro), 734
SPI_TRANS_MULTILINE_CMD (C macro), 734
SPI_TRANS_USE_RXDATA (C macro), 733
SPI_TRANS_USE_TXDATA (C macro), 733
SPI_TRANS_VARIABLE_ADDR (C macro), 733
SPI_TRANS_VARIABLE_CMD (C macro), 733
SPI_TRANS_VARIABLE_DUMMY (C macro), 733
spi_transaction_ext_t::address_bits
(C++ member), 731
spi_transaction_ext_t::base (C++ member), 731
spi_transaction_ext_t::command_bits
(C++ member), 731
spi_transaction_ext_t::dummy_bits
(C++ member), 731
spi_transaction_t (C++ struct), 730
spi_transaction_t::addr (C++ member), 731
spi_transaction_t::cmd (C++ member), 730
spi_transaction_t::flags (C++ member), 730
spi_transaction_t::length (C++ member), 731
spi_transaction_t::rx_buffer (C++ member), 731
spi_transaction_t::rx_data (C++ member), 731
spi_transaction_t::rxlength (C++ member), 731
spi_transaction_t::tx_buffer (C++ member), 731

SPICOMMON_BUSFLAG_MASTER (C macro), 723
SPICOMMON_BUSFLAG_MISO (C macro), 724
SPICOMMON_BUSFLAG_MOSI (C macro), 724
SPICOMMON_BUSFLAG_NATIVE_PINS (C macro), 724
SPICOMMON_BUSFLAG_OCTAL (C macro), 724
SPICOMMON_BUSFLAG_QUAD (C macro), 724
SPICOMMON_BUSFLAG_SCLK (C macro), 724
SPICOMMON_BUSFLAG_SLAVE (C macro), 723
SPICOMMON_BUSFLAG_WPHD (C macro), 724

StaticRingbuffer_t (C++ type), 1573
StreamBufferCallbackFunction_t (C++ type), 1545
StreamBufferHandle_t (C++ type), 1545

SUB_OPCODE_ALU_CNT (C macro), 1576
SUB_OPCODE_ALU_IMM (C macro), 1576
SUB_OPCODE_ALU_REG (C macro), 1576
SUB_OPCODE_B (C macro), 1577
SUB_OPCODE_BS (C macro), 1577
SUB_OPCODE_BX (C macro), 1577
SUB_OPCODE_END (C macro), 1578
SUB_OPCODE_MACRO_BRANCH (C macro), 1578
SUB_OPCODE_MACRO_LABEL (C macro), 1578
SUB_OPCODE_MACRO_LABELPC (C macro), 1578
SUB_OPCODE_SLEP (C macro), 1578
SUB_OPCODE_ST (C macro), 1576
SUB_OPCODE_ST_AUTO (C macro), 1576
SUB_OPCODE_ST_OFFSET (C macro), 1576

T

taskDISABLE_INTERRUPTS (C macro), 1469
taskENABLE_INTERRUPTS (C macro), 1470
taskENTER_CRITICAL (C macro), 1469
taskENTER_CRITICAL_FROM_isr (C macro), 1469
taskENTER_CRITICAL_ISR (C macro), 1469
taskEXIT_CRITICAL (C macro), 1469
taskEXIT_CRITICAL_FROM_ISR (C macro), 1469
taskEXIT_CRITICAL_ISR (C macro), 1469

TaskHandle_t (C++ type), 1475
TaskHookFunction_t (C++ type), 1475
taskSCHEDULER_NOT_STARTED (C macro), 1470
taskSCHEDULER_RUNNING (C macro), 1470
taskSCHEDULER_SUSPENDED (C macro), 1470
TaskStatus_t (C++ type), 1475
taskVALID_CORE_ID (C macro), 1469
taskYIELD (C macro), 1469

TEMPERATURE_SENSOR_CONFIG_DEFAULT (C macro), 753
temperature_sensor_config_t (C++ struct), 752
temperature_sensor_config_t::clk_src (C++ member), 752
temperature_sensor_config_t::range_max (C++ member), 752
temperature_sensor_config_t::range_min (C++ member), 752
temperature_sensor_disable (C++ function), 752
temperature_sensor_enable (C++ function), 752
temperature_sensor_get_celsius (C++ function), 752
temperature_sensor_handle_t (C++ type), 753
temperature_sensor_install (C++ function), 751
temperature_sensor_uninstall (C++ function), 752

TimerCallbackFunction_t (C++ type), 1527
TimerHandle_t (C++ type), 1527
tls_keep_alive_cfg (C++ struct), 119
tls_keep_alive_cfg::keep_alive_count (C++ member), 119
tls_keep_alive_cfg::keep_alive_enable (C++ member), 119
tls_keep_alive_cfg::keep_alive_idle (C++ member), 119
tls_keep_alive_cfg::keep_alive_interval (C++ member), 119
tls_keep_alive_cfg::keep_alive_callback (C++ member), 124
tlsDeleteCallbackFunction_t (C++ type), 1582
topic_t (C++ struct), 105
topic_t::filter (C++ member), 105
topic_t::qos (C++ member), 105
touch_button_callback_t (C++ type), 803
touch_button_config_t (C++ struct), 803
touch_button_config_t::channel_num (C++ member), 803
touch_button_config_t::channel_sens (C++ member), 803
touch_button_create (C++ function), 801
touch_elem_sleep_config_t::sample_count (C++ member), 798

**touch_elem_sleep_config_t::sleep_cycle (C++ member), 798**

touch_filter_config::smh_lvl (C++ member), 773

**touch_filter_config_t (C++ type), 774**
touch_filter_mode_t (C++ enum), 780

**touch_matrix_callback_t (C++ struct), 798**
touch_low_volt_t::TOUCH_LVOLT_MAX (C++ member), 795
touch_low_volt_t::TOUCH_LVOLT_KEEP (C++ member), 795
touch_low_volt_t::TOUCH_LVOLT_0V8 (C++ member), 795
touch_low_volt_t::TOUCH_LVOLT_0V7 (C++ member), 795
touch_low_volt_t::TOUCH_LVOLT_0V6 (C++ member), 795
touch_low_volt_t::TOUCH_LVOLT_0V5 (C++ member), 795

**touch_high_volt_t (C++ struct), 798**
touch_high_volt_t::TOUCH_HVOLT_MAX (C++ member), 795
touch_high_volt_t::TOUCH_HVOLT_KEEP (C++ member), 795
touch_high_volt_t::TOUCH_HVOLT_2V6 (C++ member), 795
touch_high_volt_t::TOUCH_HVOLT_2V5 (C++ member), 795
touch_high_volt_t::TOUCH_HVOLT_2V4 (C++ member), 795

**touch_fsm_mode_t (C++ enum), 777**
touch_fsm_mode_t::TOUCH_FSM_MODE_MAX (C++ member), 795

touch_fsm_mode_t::TOUCH_FSM_MODE_TIMER (C++ member), 795

**touch_filter_config (C++ struct), 772**
touch_filter_config::debounce_cnt (C++ member), 772

**touch_filter_config::jitter_step (C++ member), 772**
touch_filter_config::mode (C++ member), 772

touch_filter_config::noise_thr (C++ type), 812

**Index**
touch_matrix_config_t (C++ struct), 810

touch_matrix_config_t::x_channel_array (C++ member), 810
touch_matrix_config_t::x_channel_num (C++ member), 811
touch_matrix_config_t::x_sensitivity_array (C++ member), 811
touch_matrix_subscribe_event (C++ function), 814
touch_matrix_set_longpress (C++ function), 818
touch_matrix_set_dispatch_method (C++ function), 819
touch_matrix_set_callback (C++ function), 821
touch_matrix_position_t::y_axis (C++ member), 823
touch_matrix_position_t::x_axis (C++ member), 825
touch_matrix_position_t::index (C++ member), 828
touch_matrix_message_t::position (C++ member), 830
touch_matrix_message_t::event (C++ member), 832
touch_matrix_message_t::position (C++ member), 832
touch_matrix_message_t::position (C++ member), 833
touch_matrix_message_t::position (C++ member), 833
touch_matrix_message_t::position (C++ member), 834
touch_matrix_message_t::position (C++ member), 834
touch_matrix_message_t::event (C++ member), 836
touch_matrix_message_t::event (C++ member), 836

TOUCH_NOISE_THR_MAX (C macro), 774

TOUCH_PAD_ATTEN_VOLTAGE_THRESHOLD (C macro), 773

TOUCH_PAD_BIT_MASK_ALL (C macro), 773

TOUCH_PAD_BIT_MASK_MAX (C macro), 773
touch_pad_clear_channel_mask (C++ function), 759
touch_pad_clear_status (C++ function), 769
touch_pad_config (C++ function), 760
touch_pad_type_t (C++ enum), 780
touch_pad_type_t::TOUCH_PAD_CONN_GND (C++ enumerator), 780
touch_pad_type_t::TOUCH_PAD_CONN_HIZ (C++ enumerator), 780
touch_pad_type_t::TOUCH_PAD_CONN_MAX (C++ enumerator), 780
touch_pad_deinit (C++ function), 768
touch_pad_denoise (C++ struct), 772
touch_pad_denoise::cap_level (C++ member), 732
touch_pad_denoise::grade (C++ member), 732
touch_pad_denoise_cap_t (C++ enum), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L0 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L1 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L2 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L3 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L4 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L5 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L6 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_L7 (C++ enumerator), 779
touch_pad_denoise_cap_t::TOUCH_PAD_DENOISE_CAP_MAX (C++ enumerator), 779
touch_pad_denoise_disable (C++ function), 763
touch_pad_denoise_enable (C++ function), 763
touch_pad_denoise_get_config (C++ function), 763
touch_pad_denoise_grade_t (C++ enum), 778
touch_pad_denoise_grade_t::TOUCH_PAD_DENOISE_BIT1 (C++ enumerator), 778
touch_pad_denoise_grade_t::TOUCH_PAD_DENOISE_BIT1 (C++ enumerator), 778
touch_pad_denoise_grade_t::TOUCH_PAD_DENOISE_BIT4 (C++ enumerator), 779
touch_pad_denoise_grade_t::TOUCH_PAD_DENOISE_BIT8 (C++ enumerator), 779
touch_pad_denoise_grade_t::TOUCH_PAD_DENOISE_MAX (C++ enumerator), 779
touch_pad_denoise_read_data (C++ function), 760
touch_pad_intr_mask_t::TOUCH_PAD_INTR_MASK_ALL (C++ enum), 774

touch_pad_intr_mask_t::TOUCH_PAD_INTR_MASK_ALL (C++ type), 774

touch_pad_intr_disable (C++ function), 762

touch_pad_intr_enable (C++ function), 762

touch_pad_intr_get_config (C++ function), 762

touch_pad_intr_read_smooth (C++ function), 762

touch_pad_intr_set_config (C++ function), 762

touch_pad_fsm_start (C++ function), 756

touch_pad_fsm_stop (C++ function), 757

touch_pad_get_channel_mask (C++ function), 759

touch_pad_get_charge_discharge_times (C++ function), 757

touch_pad_get_cnt_mode (C++ function), 769

touch_pad_get_current_meas_channel (C++ function), 760

touch_pad_get_fsm_mode (C++ function), 769

touch_pad_get_idle_channel_connect (C++ function), 758

touch_pad_get_meas_time (C++ function), 758

touch_pad_get_measurement_interval (C++ function), 757

touch_pad_get_status (C++ function), 770

touch_pad_get_thresh (C++ function), 759

touch_pad_get_voltage (C++ function), 768

touch_pad_get_wakeup_status (C++ function), 769

touch_pad_get_idle_channel_connect (C++ function), 770

touch_pad_get_measurement_cycle (C++ function), 770

touch_pad_get_measurement_interval (C++ function), 770

touch_pad_get_status (C++ function), 770

touch_pad_get_threshold (C++ function), 770

touch_pad_get_voltage (C++ function), 770

touch_pad_get_wakeup_status (C++ function), 770

touch_pad_set_meas_time (C++ function), 770

touch_pad_set_idle_channel_connect (C++ function), 770

touch_pad_set_cnt_mode (C++ function), 770

touch_pad_set_channel_mask (C++ function), 770

touch_pad_set_fsm_mode (C++ function), 770

touch_pad_set_idle_channel_connect (C++ function), 770

touch_pad_set_meas_time (C++ function), 770

Espressif Systems

2298

Submit Document Feedback
touch_pad_t::TOUCH_PAD_NUM0

(C++ function), 775

Touch Pad Drivers (C++ function), 774

Touch Pad Drivers (C++ member), 774

Touch Pad Drivers (C++ struct), 773

Touch Pad Sleep Channel Enable (C++ function), 765

Touch Pad Sleep Channel Get Info (C++ function), 765

Touch Pad Sleep Channel Read Benchmark (C++ function), 766

Touch Pad Sleep Channel Read Data (C++ function), 766

Touch Pad Sleep Channel Read Proximity (C++ function), 767

Touch Pad Sleep Channel Read Smooth (C++ function), 766

Touch Pad Sleep Channel Set Work Time (C++ function), 767

Touch Pad Sleep Channel T (C++ struct), 773

Touch Pad Sleep Channel T::en_proximity (C++ member), 773

Touch Pad Sleep Channel T::touch_num (C++ member), 773

TOUCH_PAD_SLEEP_CYCLE_DEFAULT (C macro), 773

Touch Pad Sleep Get Threshold (C++ function), 766

Touch Pad Sleep Set Threshold (C++ function), 766

TOUCH_PAD_SLOPE_DEFAULT (C macro), 773

Touch Pad Sw Start (C++ function), 775

Touch T (C++ enum), 774

Touch T::TOUCH_PAD_MAX (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM0 (C++ enumerator), 774

Touch T::TOUCH_PAD_NUM1 (C++ enumerator), 774

Touch T::TOUCH_PAD_NUM10 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM11 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM12 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM13 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM14 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM2 (C++ enumerator), 774

Touch T::TOUCH_PAD_NUM3 (C++ enumerator), 774

Touch T::TOUCH_PAD_NUM4 (C++ enumerator), 774

Touch T::TOUCH_PAD_NUM5 (C++ enumerator), 774

Touch T::TOUCH_PAD_NUM6 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM7 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM8 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM9 (C++ enumerator), 775

Touch T::TOUCH_PAD_NUM11 (C++ struct), 773

Touch T::TOUCH_PAD_NUM12 (C++ struct), 773

Touch T::TOUCH_PAD_NUM13 (C++ struct), 773

Touch T::TOUCH_PAD_NUM14 (C++ struct), 773

TOUCH_PROXIMITY_MEAS_NUM_MAX (C macro), 774

Touch Slider Callback T (C++ type), 807

Touch Slider Config T (C++ struct), 806

Touch Slider Config T::channel_array (C++ member), 807

Touch Slider Config T::channel_num (C++ member), 807

Touch Slider Config T::position_range (C++ member), 807

Submit Document Feedback
touch_slider_config_t::sensitivity_array (C++ member), 807

C++ member, 807

touch_slider_create (C++ function), 804

C++ function, 804

touch_slider_delete (C++ function), 805

C++ function, 805

touch_slider_event_t (C++ enum), 807

C++ enum), 807

touch_slider_event_t::TOUCH_SLIDER_EVT_MAX (C++ enumerator), 781

C++ enumerator, 781

touch_slider_global_config_t (C++ struct), 806

C++ struct, 806

touch_slider_global_config_t::benchmark_update_time (C++ member), 806

C++ member, 806

touch_slider_global_config_t::calculate_channel_count (C++ member), 806

C++ member, 806

touch_slider_global_config_t::filter_reset_time (C++ member), 806

C++ member, 806

touch_slider_global_config_t::position_filter_reset_time (C++ member), 806

C++ member, 806

touch_slider_global_config_t::position_threshold (C++ member), 806

C++ member, 806

TOUCH_SLIDER_GLOBAL_DEFAULT_CONFIG (C macro), 807

C macro, 807

touch_slider_handle_t (C++ type), 807

C++ type, 807

touch_slider_install (C++ function), 804

C++ function, 804

touch_slider_message_t (C++ function), 807

C++ function, 807

touch_slider_message_t::event (C++ member), 807

C++ member, 807

touch_slider_message_t::position (C++ member), 807

C++ member, 807

touch_slider_position_t (C++ type), 807

C++ type, 807

touch_slider_set_callback (C++ function), 805

C++ function, 805

touch_slider_set_dispatch_method (C++ function), 805

C++ function, 805

touch_slider_subscribe_event (C++ function), 805

C++ function, 805

touch_slider_uninstall (C++ function), 804

C++ function, 804

touch_smooth_mode_t (C++ enum), 781

C++ enum), 781

touch_smooth_mode_t::TOUCH_PAD_SMOOTH_IIR_4 (C++ enumerator), 781

C++ enumerator, 781

touch_smooth_mode_t::TOUCH_PAD_SMOOTH_IIR_2 (C++ enumerator), 781

C++ enumerator, 781

touch_smooth_mode_t::TOUCH_PAD_SMOOTH_IIR_8 (C++ enumerator), 781

C++ enumerator, 781

touch_smooth_mode_t::TOUCH_PAD_SMOOTH_OFF (C++ enumerator), 781

C++ enumerator, 781

touch_smooth_mode_t::TOUCH_PAD_SMOOTH_MAX (C++ enumerator), 781

C++ enumerator, 781

touch_smooth_mode_t::TOUCH_SLIDER_EVT_MAX (C++ function), 807

C++ function, 807

C++ function, 807
uart_event_type_t::UART_DATA
uart_event_type_t::UART_BUFFER_FULL
uart_event_type_t::UART_BREAK
uart_event_type_t::UART_FRAME_ERR
uart_event_type_t::UART_WAKEUP

twai_timing_config_t::tseg_2 (C++ member), 824
twai_transmit (C++ function), 828
twai_transmit_v2 (C++ function), 828

U
uart_at_cmd_t (C++ struct), 854
uart_at_cmd_t::char_num (C++ member), 855
uart_at_cmd_t::cmd_char (C++ member), 855
uart_at_cmd_t::gap_tout (C++ member), 855
uart_at_cmd_t::post_idle (C++ member), 855
uart_at_cmd_t::pre_idle (C++ member), 855
UART_BITRATE_MAX (C macro), 853
uart_clear_intr_status (C++ function), 844
uart_config_t (C++ struct), 852
uart_config_t::baud_rate (C++ member), 852
uart_config_t::data_bits (C++ member), 852
uart_config_t::flow_ctrl (C++ member), 852
uart_config_t::parity (C++ member), 852
uart_config_t::rx_flow_ctrl_thresh (C++ member), 852
uart_config_t::source_clk (C++ member), 852
uart_config_t::stop_bits (C++ member), 852
UART_CTS_GPIO16_DIRECT_CHANNEL (C macro), 859
UART_CTS_GPIO20_DIRECT_CHANNEL (C macro), 859
uart_disable_intr_mask (C++ function), 844
uart_disable_pattern_det_intr (C++ function), 848
uart_disable_rx_intr (C++ function), 844
uart_disable_tx_intr (C++ function), 844
uart_driver_delete (C++ function), 841
uart_driver_install (C++ function), 841
uart_enable_intr_mask (C++ function), 844
uart_enable_pattern_det_baud_intr (C++ function), 848
uart_enable_rx_intr (C++ function), 844
uart_enable_tx_intr (C++ function), 844
uart_event_t (C++ struct), 853
uart_event_t::size (C++ member), 853
uart_event_t::timeout_flag (C++ member), 853
uart_event_t::type (C++ member), 853
uart_event_type_t (C++ enum), 854
uart_event_type_t::UART_BREAK (C++ enumerator), 854
uart_event_type_t::UART_EVENT_MAX (C++ enumerator), 854
uart_event_type_t::UART_FIFO_OVF (C++ enumerator), 854
uart_event_type_t::UART_FRAME_ERR (C++ enumerator), 854
uart_event_type_t::UART_PARITY_ERR (C++ enumerator), 854
uart_event_type_t::UART_PATTERN_DET (C++ enumerator), 854
uart_event_type_t::UART_WAKEUP (C++ enumerator), 854
UART_FIFO_LEN (C macro), 853
uart_flush (C++ function), 847
uart_flush_input (C++ function), 847
uart_get_baudrate (C++ function), 843
uart_get_buffered_data_len (C++ function), 848
uart_get_collision_flag (C++ function), 850
uart_get_hw_flowctrl (C++ function), 843
uart_get_hw_flowctrl (C++ function), 842
uart_get_sclk_freq (C++ function), 842
uart_get_stop_bits (C++ function), 842
uart_get_tx_buffer_free_size (C++ function), 848
uart_get_wakeup_threshold (C++ function), 851
uart_get_word_length (C++ function), 842
UART_GPIO15_DIRECT_CHANNEL (C macro), 859
UART_GPIO16_DIRECT_CHANNEL (C macro), 859
UART_GPIO17_DIRECT_CHANNEL (C macro), 859
UART_GPIO18_DIRECT_CHANNEL (C macro), 859
UART_GPIO19_DIRECT_CHANNEL (C macro), 859
UART_GPIO20_DIRECT_CHANNEL (C macro), 859
UART_GPIO43_DIRECT_CHANNEL (C macro), 859
UART_GPIO44_DIRECT_CHANNEL (C macro), 859
UART_HW_FIFO_LEN (C macro), 853
uart_hw_flowcontrol_t (C++ enum), 857
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_CTS (C++ enumerator), 857
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_CTS RTS (C++ enumerator), 857
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_DISABLE (C++ enumerator), 857
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_MAX (C++ enumerator), 857
uart_hw_flowcontrol_t::UART_HW_FLOWCTRL_RTS (C++ enumerator), 857
uart_intr_config (C++ function), 846
uart_intr_config_t (C++ struct), 852
uart_intr_config_t::intr_enable_mask (C++ member), 853
uart_intr_config_t::rx_timeout_thresh (C++ member), 853
uart_intr_config_t::tx_fifofull_thresh (C++ member), 853
uart_intr_config_t::txfifo_empty_intr_thresh macro, 859
(C++ member), 853
uart_is_driver_installed (C++ function), 841
uart_isr_handle_t (C++ type), 854
uart_mode_t (C++ enum), 856
uart_mode_t::UART_MODE_IRDA (C++ enumerator), 856
uart_mode_t::UART_MODE_RS485_APP_CTRL (C++ enumerator), 856
uart_mode_t::UART_MODE_RS485_COLLISION_DETECT (C++ enumerator), 856
uart_mode_t::UART_MODE_RS485_HALF_DUPLEX (C++ enumerator), 856
uart_mode_t::UART_MODE_UART (C++ enumerator), 856
UART_NUM_0_CTS_DIRECT_GPIO_NUM macro, 859
UART_NUM_0_CTS_DIRECT_GPIO_NUM macro, 859
UART_NUM_0_RXD_DIRECT_GPIO_NUM macro, 859
UART_NUM_0_TXD_DIRECT_GPIO_NUM macro, 859
UART_NUM_1_CTS_DIRECT_GPIO_NUM macro, 859
UART_NUM_1_CTS_DIRECT_GPIO_NUM macro, 859
UART_NUM_1_RTS_DIRECT_GPIO_NUM macro, 859
UART_NUM_1_TXD_DIRECT_GPIO_NUM macro, 859
uart_param_config (C++ function), 846
uart_parity_t (C++ enum), 857
uart_parity_t::UART_PARITY_DISABLE (C++ enumerator), 857
uart_parity_t::UART_PARITY_EVEN (C++ enumerator), 857
uart_parity_t::UART_PARITY_ODD (C++ enumerator), 857
uart_pattern_get_pos (C++ function), 849
uart_pattern_pcp_pos (C++ function), 849
uart_pattern_queue_reset (C++ function), 849
UART_PIN_NO_CHANGE (C macro), 853
uart_port_t (C++ enum), 855
uart_port_t::UART_NUM_0 (C++ enumerator), 855
uart_port_t::UART_NUM_1 (C++ enumerator), 855
uart_read_bytes (C++ function), 847
UART_RTS_GPIO15_DIRECT_CHANNEL macro, 859
UART_RTS_GPIO19_DIRECT_CHANNEL macro, 859
UART_RXD_GPIO18_DIRECT_CHANNEL UART_RXD_GPIO44_DIRECT_CHANNEL (C macro), 859
ten clk_t (C++ type), 855
uart_set_always_rx_timeout (C++ function), 852
uart_set_baudrate (C++ function), 843
uart_set_dtr (C++ function), 845
uart_set_hw_flow_ctl (C++ function), 843
uart_set_line_inverse (C++ function), 843
detach_device_loop_back (C++ function), 851
uart_set_mode (C++ function), 849
uart_set_rts (C++ function), 842
uart_set_pin (C++ function), 845
uart_set_rts (C++ function), 845
uart_set_rx_full_threshold (C++ function), 850
uart_set_rx_timeout (C++ function), 850
uart_set_stop_bits (C++ function), 842
uart_set_sw_flow_ctl (C++ function), 843
uart_set_tx_empty_threshold (C++ function), 850
uart_set_rx_empty_threshold (C++ function), 846
uart_set_word_length (C++ function), 841
uart_signal_inv_t::UART_SIGNAL_DTR_INV (C++ enumerator), 858
uart_signal_inv_t::UART_SIGNAL_DSR_INV (C++ enumerator), 858
uart_signal_inv_t::UART_SIGNAL_DSR_INV (C++ enumerator), 858
uart_signal_inv_t::UART_SIGNAL_IRDA_RX_INV (C++ enumerator), 858
uart_signal_inv_t::UART_SIGNAL_IRDA_RX_INV (C++ enumerator), 858
uart_signal_inv_t::UART_SIGNAL_IRDA_TX_INV (C++ enumerator), 857
uart_signal_inv_t::UART_SIGNAL_IRDA_TX_INV (C++ enumerator), 857
uart_stop_bits_t (C++ enum), 856
uart_stop_bits_t::UART_STOP_BITS_1 (C++ enum), 855
uart_stop_bits_t::UART_STOP_BITS_1_5 (C++ enum), 857
uart_stop_bits_t::UART_STOP_BITS_2 (C++ enum), 857
uart_stop_bits_t::UART_STOP_BITS_MAX (C++ enum), 857
uart_sw_flowctrl_t::xoff_char (C++ member), 855
uart_sw_flowctrl_t::xoff_thrd (C++ member), 855
Index

uart_sw_flowctrl_t::xon_char (C++ member), 855
uart_sw_flowctrl_t::xon_thrd (C++ member), 855
uart_tx_chars (C++ function), 846
UART_TXD_GPIO17_DIRECT_CHANNEL_MACRO, 859
UART_TXD_GPIO43_DIRECT_CHANNEL_MACRO, 859
uart_wait_tx_done (C++ function), 846
uart_wait_tx_idle_polling (C++ function),
851
uart_word_length_t (C++ enum), 856
uart_word_length_t::UART_DATA_5_BITS
(C++ enumerator), 856
uart_word_length_t::UART_DATA_6_BITS
(C++ enumerator), 856
uart_word_length_t::UART_DATA_7_BITS
(C++ enumerator), 856
uart_word_length_t::UART_DATA_8_BITS
(C++ enumerator), 856
uart_word_length_t::UART_DATA_BITS_MAX
(C++ enumerator), 856
uart_write_bytes (C++ function), 846
uart_write_bytes_with_break (C++ function),
847
ulp_insn (C++ union), 1751
ulp_insn::adc (C++ member), 1754
ulp_insn::addr (C++ member), 1753
ulp_insn::alu_cnt (C++ member), 1754
ulp_insn::alu_imm (C++ member), 1753
ulp_insn::alu_reg (C++ member), 1753
ulp_insn::*b (C++ member), 1753
ulp_insn::*bx (C++ member), 1753
ulp_insn::cmp (C++ member), 1753
ulp_insn::cycles (C++ member), 1751
ulp_insn::data (C++ member), 1754
ulp_insn::delay (C++ member), 1751
ulp_insn::dreg (C++ member), 1752
ulp_insn::*end (C++ member), 1755
ulp_insn::*halt (C++ member), 1753
ulp_insn::*high (C++ member), 1754
ulp_insn::*high_bits (C++ member), 1755
ulp_insn::*i2c (C++ member), 1755
ulp_insn::*i2c_addr (C++ member), 1754
ulp_insn::*i2c_sel (C++ member), 1755
ulp_insn::*imm (C++ member), 1753
ulp_insn::*label (C++ member), 1752
ulp_insn::*ld (C++ member), 1753
ulp_insn::*low (C++ member), 1754
ulp_insn::*low_bits_t (C++ member), 1754
ulp_insn::*macro (C++ member), 1755
ulp_insn::*mux (C++ member), 1754
ulp_insn::*offset (C++ member), 1752
ulp_insn::*opcode_t (C++ member), 1751
ulp_insn::*periph_sel (C++ member), 1754
ulp_insn::rd_reg (C++ member), 1754
ulp_insn::rd_upper (C++ member), 1753
ulp_insn::reg (C++ member), 1753
ulp_insn::reserved (C++ member), 1754
ulp_insn::rw (C++ member), 1755
ulp_insn::sar_sel (C++ member), 1754
ulp_insn::sel (C++ member), 1753
ulp_insn::sign (C++ member), 1753
ulp_insn::sreg (C++ member), 1752
ulp_insn::st (C++ member), 1752
ulp_insn::sub_opcode (C++ member), 1752
ulp_insn::treg (C++ member), 1753
ulp_insn::tsens (C++ member), 1754
ulp_insn::type (C++ member), 1753
ulp_insn::unused (C++ member), 1751
ulp_insn::unused1 (C++ member), 1752
ulp_insn::unused2 (C++ member), 1752
ulp_insn::unused3 (C++ member), 1754
ulp_insn::upper (C++ member), 1752
ulp_insn::wait_delay (C++ member), 1754
ulp_insn::waitup (C++ member), 1755
ulp_insn::wr_reg (C++ member), 1754
ulp_insn::wr_way (C++ member), 1752
ulp_insn::w (C++ function), 1770
ulp_isr_deregister (C++ function), 1768
ulp_isr_register (C++ function), 1768
ulp_load_binary (C++ function), 1769
ulp_process_macros_and_load (C++ function),
1769
ulp_riscv_cfg_t (C++ struct), 1777
ulp_riscv_cfg_t::wakeup_source (C++
member), 1778
ulp_riscv_config_and_run (C++ function),
1776
ULP_RISCV_DEFAULT_CONFIG (C macro), 1778
ulp_riscv_halt (C++ function), 1777
ulp_riscv_i2c_cfg_t (C++ struct), 1781
ulp_riscv_i2c_cfg_t::i2c_pin_cfg (C++
member), 1781
ulp_riscv_i2c_cfg_t::i2c_timing_cfg
(C++ member), 1781
ULP_RISCV_I2C_DEFAULT_CONFIG (C macro),
1782
ULP_RISCV_I2C_DEFAULT_GPIO_CONFIG (C
macro), 1782
ulp_riscv_i2c_master_init (C++ function),
1780
ulp_riscv_i2c_master_read_from_device
(C++ function), 1780
ulp_riscv_i2c_master_set_slave_addr
(C++ function), 1780
ulp_riscv_i2c_master_set_slave_reg_addr
(C++ function), 1780
ulp_riscv_i2c_master_write_to_device
(C++ function), 1780
ulp_riscv_i2c_pin_cfg_t (C++ struct), 1780
Index

ulp_riscv_i2c_pin_cfg_t::scl_io_num
tion), 1466
(C++ member), 1781
ulTaskGetIdleRunTimeCounter (C++ funculp_riscv_i2c_pin_cfg_t::scl_pullup_en
tion), 1463
(C++ member), 1781
ulTaskGetIdleRunTimePercent (C++ funculp_riscv_i2c_pin_cfg_t::sda_io_num
tion), 1463
(C++ member), 1780
ulTaskNotifyTakeIndexed (C macro), 1473
ulp_riscv_i2c_pin_cfg_t::sda_pullup_en ulTaskNotifyValueClear (C macro), 1475
(C++ member), 1781
ulTaskNotifyValueClearIndexed (C macro),
ULP_RISCV_I2C_STANDARD_MODE_CONFIG (C
1475
macro), 1782
USB_B_DESCRIPTOR_TYPE_BOS (C macro), 896
ulp_riscv_i2c_timing_cfg_t (C++ struct), USB_B_DESCRIPTOR_TYPE_CONFIGURATION (C
1781
macro), 896
ulp_riscv_i2c_timing_cfg_t::i2c_trans_timeout
USB_B_DESCRIPTOR_TYPE_CS_RADIO_CONTROL
(C++ member), 1781
(C macro), 897
ulp_riscv_i2c_timing_cfg_t::scl_high_period
USB_B_DESCRIPTOR_TYPE_DEBUG (C macro),
896
(C++ member), 1781
ulp_riscv_i2c_timing_cfg_t::scl_low_period
USB_B_DESCRIPTOR_TYPE_DEVICE (C macro),
896
(C++ member), 1781
ulp_riscv_i2c_timing_cfg_t::scl_start_period
USB_B_DESCRIPTOR_TYPE_DEVICE_CAPABILITY
(C++ member), 1781
(C macro), 896
ulp_riscv_i2c_timing_cfg_t::scl_stop_period
USB_B_DESCRIPTOR_TYPE_DEVICE_QUALIFIER
(C++ member), 1781
(C macro), 896
ulp_riscv_i2c_timing_cfg_t::sda_duty_period
USB_B_DESCRIPTOR_TYPE_ENCRYPTION_TYPE
(C++ member), 1781
(C macro), 896
ulp_riscv_isr_deregister (C++ function), USB_B_DESCRIPTOR_TYPE_ENDPOINT
(C
macro), 896
1776
ulp_riscv_isr_register (C++ function), 1776 USB_B_DESCRIPTOR_TYPE_INTERFACE
(C
macro), 896
ulp_riscv_load_binary (C++ function), 1777
ulp_riscv_lock_acquire (C++ function), 1779 USB_B_DESCRIPTOR_TYPE_INTERFACE_ASSOCIATION
ulp_riscv_lock_release (C++ function), 1779
(C macro), 896
ulp_riscv_lock_t (C++ struct), 1778
USB_B_DESCRIPTOR_TYPE_INTERFACE_POWER
ulp_riscv_lock_t::critical_section_flag_main_cpu
(C macro), 896
(C++ member), 1778
USB_B_DESCRIPTOR_TYPE_KEY (C macro), 896
ulp_riscv_lock_t::critical_section_flag_ulp
USB_B_DESCRIPTOR_TYPE_OTG (C macro), 896
(C++ member), 1778
USB_B_DESCRIPTOR_TYPE_OTHER_SPEED_CONFIGURATION
(C macro), 896
ulp_riscv_lock_t::turn (C++ member), 1778
ulp_riscv_lock_turn_t (C++ enum), 1779
USB_B_DESCRIPTOR_TYPE_PIPE_USAGE
(C
ulp_riscv_lock_turn_t::ULP_RISCV_LOCK_TURN_MAIN_CPU
macro), 897
(C++ enumerator), 1779
USB_B_DESCRIPTOR_TYPE_RPIPE (C macro),
ulp_riscv_lock_turn_t::ULP_RISCV_LOCK_TURN_ULP897
(C++ enumerator), 1779
USB_B_DESCRIPTOR_TYPE_SECURITY
(C
ulp_riscv_reset (C++ function), 1777
macro), 896
ulp_riscv_run (C++ function), 1777
USB_B_DESCRIPTOR_TYPE_STRING (C macro),
ULP_RISCV_SW_INT (C macro), 1778
896
ulp_riscv_timer_resume (C++ function), 1777 USB_B_DESCRIPTOR_TYPE_WIRE_ADAPTER (C
ulp_riscv_timer_stop (C++ function), 1777
macro), 896
ULP_RISCV_TRAP_INT (C macro), 1778
USB_B_DESCRIPTOR_TYPE_WIRELESS_ENDPOINT_COMP
ulp_riscv_wakeup_source_t (C++ enum),
(C macro), 896
1778
USB_B_ENDPOINT_ADDRESS_EP_DIR_MASK (C
ulp_riscv_wakeup_source_t::ULP_RISCV_WAKEUP_SOURCE_GPIO
macro), 900
(C++ enumerator), 1778
USB_B_ENDPOINT_ADDRESS_EP_NUM_MASK (C
ulp_riscv_wakeup_source_t::ULP_RISCV_WAKEUP_SOURCE_TIMER
macro), 900
USB_B_REQUEST_CLEAR_FEATURE (C macro),
(C++ enumerator), 1778
ulp_run (C++ function), 1769
897
ulp_set_wakeup_period (C++ function), 1770
USB_B_REQUEST_GET_CONFIGURATION
(C
ulp_timer_resume (C++ function), 1771
macro), 898
ulp_timer_stop (C++ function), 1771
USB_B_REQUEST_GET_DESCRIPTOR (C macro),
ulTaskGenericNotifyValueClear (C++ func897

Espressif Systems

2305
Submit Document Feedback

Release master


USB_B_REQUEST_GET_INTERFACE (C macro), 898
USB_B_REQUEST_GET_STATUS (C macro), 897
USB_B_REQUEST_SET_ADDRESS (C macro), 897
USB_B_REQUEST_SET_CONFIGURATION (C macro), 898
USB_B_REQUEST_SET_DESCRIPTOR (C macro), 898
USB_B_REQUEST_SET_FEATURE (C macro), 897
USB_B_REQUEST_SET_INTERFACE (C macro), 897
USB_B_REQUEST_SYNCH_FRAME (C macro), 898
USB_B_ATTRIBUTES_BATTERY (C macro), 900
USB_B_ATTRIBUTES_SELFPOWER (C macro), 900
USB_B_ATTRIBUTES_SYNC_ADAPTIVE (C macro), 901
USB_B_ATTRIBUTES_SYNC_ASYNC (C macro), 901
USB_B_ATTRIBUTES_SYNC_NONE (C macro), 901
USB_B_ATTRIBUTES_SYNC_SYNC (C macro), 901
USB_B_ATTRIBUTES_SYNTYPE_MASK (C macro), 901
USB_B_ATTRIBUTES_USAGE_DATA (C macro), 901
USB_B_ATTRIBUTES_USAGE_FEEDBACK (C macro), 901
USB_B_ATTRIBUTES_USAGE_IMPLICIT_FB (C macro), 901
USB_B_ATTRIBUTES_USAGETYPE_MASK (C macro), 901
USB_B_ATTRIBUTES_WAKEUP (C macro), 900
USB_B_ATTRIBUTES_XFER_BULK (C macro), 901
USB_B_ATTRIBUTES_XFER_CONTROL (C macro), 900
USB_B_ATTRIBUTES_XFER_INT (C macro), 901
USB_B_ATTRIBUTES_XFER_ISOC (C macro), 901
USB_B_ATTRIBUTES_XFERTYPE_MASK (C macro), 900
USB_B_REQUEST_TYPE_DIR_IN (C macro), 897
USB_B_REQUEST_TYPE_DIR_OUT (C macro), 897
USB_B_REQUEST_TYPE_RECIP_DEVICE (C macro), 897
USB_B_REQUEST_TYPE_RECIP_ENDPOINT (C macro), 897
USB_B_REQUEST_TYPE_RECIP_INTERFACE (C macro), 897
USB_B_REQUEST_TYPE_RECIP_MASK (C macro), 897
USB_B_REQUEST_TYPE_RECIP_OTHER (C macro), 897
USB_B_REQUEST_TYPE_TYPE_CLASS (C macro), 897
USB_B_REQUEST_TYPE_TYPE_MASK (C macro), 897
USB_B_REQUEST_TYPE_TYPE_STANDARD (C macro), 897
USB_B_REQUEST_TYPE_TYPE_RESERVED (C macro), 897
USB_B_REQUEST_TYPE_TYPE_VENDOR (C macro), 897
USB_CLASS_APP_SPEC (C macro), 900
USB_CLASS_AUDIO (C macro), 899
USB_CLASS_AUDIO_VIDEO (C macro), 899
USB_CLASS_BATTERY (C macro), 899
USB_CLASS_BILLBOARD (C macro), 899
USB_CLASS_CDC_DATA (C macro), 899
USB_CLASS_COMM (C macro), 899
USB_CLASS_CONTENT_SEC (C macro), 899
USB_CLASS_CSCID (C macro), 899
USB_CLASS_CDC_DATA (C macro), 899
USB_CLASS_HUB (C macro), 899
USB_CLASS_MASS_STORAGE (C macro), 899
USB_CLASS_MISC (C macro), 900
USB_CLASS_PER_INTERFACE (C macro), 899
USB_CLASS_PERSONAL_HEALTHCARE (C macro), 899
USB_CLASS_PHYSICAL (C macro), 899
USB_CLASS_PRINTER (C macro), 899
USB_CLASS_STILL_IMAGE (C macro), 899
USB_CLASS_USB_TYPE_C_BRIDGE (C macro), 899
USB_CLASS_VID (C macro), 899
USB_CLASS_VIDEO (C macro), 899
USB_CLASS_WIRELESS_CONTROLLER (C macro), 899
USB_CONFIG_DESC_SIZE (C macro), 900
usb_config_desc_t (C++ union), 892
usb_config_desc_t::bConfigurationValue (C++ member), 893
usb_config_desc_t::bDescriptorType (C++ member), 892
usb_config_desc_t::bLength (C++ member), 892
usb_config_desc_t::bmAttributes (C++ member), 893
usb_config_desc_t::bMaxPower (C++ member), 893
usb_config_desc_t::iConfiguration (C++ member), 893
usb_config_desc_t::USB_DESC_ATTRIB (C++ member), 893
usb_config_desc_t::wTotalLength (C++ member), 892
usb_Device_desc_t (C++ union), 891
Index

usb_device_desc_t::bcdDevice (C++ member), 892
usb_device_desc_t::bcdUSB (C++ member), 891
usb_device_desc_t::bDescriptorType (C++ member), 891
usb_device_desc_t::bDeviceClass (C++ member), 891
usb_device_desc_t::bDeviceProtocol (C++ member), 891
usb_device_desc_t::bDeviceSubClass (C++ member), 891
usb_device_desc_t::bLength (C++ member), 891
usb_device_desc_t::bMaxPacketSize0 (C++ member), 891
usb_device_desc_t::bNumConfigurations (C++ member), 892
usb_device_desc_t::idProduct (C++ member), 892
usb_device_desc_t::idVendor (C++ member), 892
usb_device_desc_t::iManufacturer (C++ member), 892
usb_device_desc_t::iProduct (C++ member), 892
usb_device_desc_t::iSerialNumber (C++ member), 892
usb_device_desc_t::USB_DESC_ATTR (C++ member), 892
usb_device_desc_t::val (C++ member), 892
usb_device_handle_t (C++ type), 888
usb_device_info_t (C++ struct), 885
usb_device_info_t::bConfigurationValue (C++ member), 886
usb_device_info_t::bMaxPacketSize0 (C++ member), 886
usb_device_info_t::dev_addr (C++ member), 885
usb_device_info_t::speed (C++ member), 885
usb_device_info_t::str_desc_manufacturer (C++ member), 886
usb_device_info_t::str_desc_product (C++ member), 886
usb_device_info_t::str_desc_serial_num (C++ member), 886
usb_device_state_t (C++ enum), 901
usb_device_state_t::USB_DEVICE_STATE_ADDRESS (C++ enumerator), 901
usb_device_state_t::USB_DEVICE_STATE_CONFIGURED (C++ enumerator), 902
usb_device_state_t::USB_DEVICE_STATE_DEFAULT (C++ enumerator), 902
usb_device_state_t::USB_DEVICE_STATE_NOT_ATTACHED (C++ enumerator), 901
usb_device_state_t::USB_DEVICE_STATE_POWERED (C++ enumerator), 902
usb_device_state_t::USB_DEVICE_STATE_SUSPENDED (C++ enumerator), 902
usb_EP_DESC_GET_EP_DIR (C macro), 901
usb_EP_DESC_GET_EP_NUM (C macro), 901
usb_EP_DESC_GET_MPS (C macro), 901
usb_EP_DESC_GET_XFERTYPE (C macro), 901
usb_EP_DESC_SIZE (C macro), 900
usb_ep_desc_t (C++ union), 894
usb_ep_desc_t::bDescriptorType (C++ member), 895
usb_ep_desc_t::bEndpointAddress (C++ member), 895
usb_ep_desc_t::bInterval (C++ member), 895
usb_ep_desc_t::bLength (C++ member), 895
usb_ep_desc_t::bmAttributes (C++ member), 895
usb_ep_desc_t::USB_DESC_ATTR (C++ member), 895
usb_ep_desc_t::val (C++ member), 895
usb_ep_desc_t::wMaxPacketSize (C++ member), 895
usb_host_client_config_t (C++ struct), 881
usb_host_client_config_t::async (C++ member), 881
usb_host_client_config_t::callback_arg (C++ member), 881
usb_host_client_config_t::client_event_callback (C++ member), 881
usb_host_client_config_t::is_synchronous (C++ member), 881
usb_host_client_config_t::max_num_event_msg (C++ member), 881
usb_host_client_deregister (C++ function), 875
usb_host_client_event_cb_t (C++ type), 882
usb_host_client_event_msg_t (C++ struct), 880
usb_host_client_event_msg_t::address (C++ member), 880
usb_host_client_event_msg_t::dev_addr (C++ member), 880
usb_host_client_event_msg_t::dev_smart (C++ member), 880
usb_host_client_event_msg_t::power (C++ member), 880
usb_host_client_event_msg_t::status (C++ member), 880
usb_host_client_event_msg_t::status (C++ member), 880
usb_host_client_event_msg_t::type (C++ member), 880
usb_host_client_handle_events (C++ function), 875
usb_host_client_handle_t (C++ type), 882
Index

usb_host_client_register (C++ function), 874
usb_host_client_unregister (C++ function), 874
usb_host_config_t (C++ struct), 881
usb_host_config_t::*intr_flags (C++ member), 881
usb_host_config_t::skip_phy_setup (C++ member), 881
usb_host_device_addr_list_fill (C++ function), 876
usb_host_device_close (C++ function), 875
usb_host_device_free_all (C++ function), 876
usb_host_device_info (C++ function), 876
usb_host_device_open (C++ function), 875
usb_host_endpoint_clear (C++ function), 878
usb_host_endpoint_flush (C++ function), 878
usb_host_endpoint_halt (C++ function), 878
usb_host_get_active_config_descriptor (C++ function), 877
usb_host_get_device_descriptor (C++ function), 877
usb_host_install (C++ function), 873
usb_host_interface_claim (C++ function), 877
usb_host_interface_release (C++ function), 878
USB_HOST_LIB_EVENT_FLAGS_ALL_FREE (C macro), 881
USB_HOST_LIB_EVENT_FLAGS_NO_CLIENTS (C macro), 881
usb_host_lib_handle_events (C++ function), 874
usb_host_lib_info (C++ function), 874
usb_host_lib_info_t (C++ struct), 880
usb_host_lib_info_t::*num_clients (C++ member), 881
usb_host_lib_info_t::*num_devices (C++ member), 881
usb_host_lib_unregister (C++ function), 874
usb_host_transfer_alloc (C++ function), 879
usb_host_transfer_free (C++ function), 879
usb_host_transfer_submit (C++ function), 879
usb_host_transfer_submit_control (C++ function), 879
usb_host_unregister (C++ function), 873
USB_IAD_DESC_SIZE (C macro), 900
usb_iad_desc_t::*bDescriptorType (C++ member), 893
usb_iad_desc_t::*bFirstInterface (C++ member), 893
usb_iad_desc_t::*bFunctionClass (C++ member), 893
usb_iad_desc_t::*bFunctionProtocol (C++ member), 893
usb_iad_desc_t::*bFunctionSubClass (C++ member), 893
usb_iad_desc_t::*bInterfaceCount (C++ member), 893
usb_iad_desc_t::*bLength (C++ member), 893
usb_iad_desc_t::*iFunction (C++ member), 893
usb_iad_desc_t::USB_DESC_ATTR (C++ member), 894
usb_iad_desc_t::*bDescriptorType (C++ member), 894
usb_iad_desc_t::*bInterfaceClass (C++ member), 894
usb_iad_desc_t::*bInterfaceNumber (C++ member), 894
usb_iad_desc_t::*bInterfaceProtocol (C++ member), 894
usb_iad_desc_t::*bInterfaceSubClass (C++ member), 894
usb_iad_desc_t::*bNumEndpoints (C++ member), 894
usb_iad_desc_t::*bNumEndpoints (C++ member), 894
usb_iad_desc_t::*iInterface (C++ member), 894
usb_iad_desc_t::USB_DESC_ATTR (C++ member), 894
usb_iad_desc_t::*val (C++ member), 904
USB_INTF_DESC_SIZE (C macro), 900
usb_intf_desc_t (C++ union), 894
usb_intf_desc_t::*bAlternateSetting (C++ member), 894
usb_intf_desc_t::*bDescriptorType (C++ member), 894
usb_intf_desc_t::*bInterfaceClass (C++ member), 894
usb_intf_desc_t::*bInterfaceNumber (C++ member), 894
usb_intf_desc_t::*bInterfaceProtocol (C++ member), 894
usb_intf_desc_t::*bInterfaceSubClass (C++ member), 894
usb_isoc_packet_desc_t::status (C++ member), 886
usb_isoc_packet_desc_t::num_bytes (C++ member), 886
usb_isoc_packet_desc_t::actual_num_bytes (C++ member), 886
usb_parse_endpoint_descriptor_by_address (C++ function), 884
usb_parse_endpoint_descriptor_by_index (C++ function), 884
usb_parse_interface_descriptor (C++ function), 883
usb_parse_interface_number_of_alternate (C++ function), 883
usb_parse_next_descriptor (C++ function), 882
usb_parse_next_descriptor_of_type (C++ function), 883
usb_print_config_descriptor (C++ function), 884
usb_print_device_descriptor (C++ function), 884
usb_print_string_descriptor (C++ function), 885
usb_round_up_to_mps (C++ function), 885

Espressif Systems

Release master

Submit Document Feedback
USB_SETUP_PACKET_INIT_GET_CONFIG (C macro), 898
USB_SETUP_PACKET_INIT_GET_CONFIG_DESC (C macro), 898
USB_SETUP_PACKET_INIT_GET_DEVICE_DESC (C macro), 898
USB_SETUP_PACKET_INIT_GET_STR_DESC (C macro), 899
USB_SETUP_PACKET_INIT_GET_ADDR (C macro), 898
USB_SETUP_PACKET_INIT_SET_CONFIG (C macro), 898
USB_SETUP_PACKET_INIT_SET_INTERFACE (C macro), 898
USB_SETUP_PACKET_SIZE (C macro), 897
usb_setup_packet_t::bRequestType (C++ member), 890
usb_setup_packet_t::bRequest (C++ member), 890
usb_setup_packet_t::val (C++ member), 890
usb_setup_packet_t::wLength (C++ member), 890
usb_setup_packet_t::wValue (C++ member), 890
usb_setup_packet_t::[anonymous] (C++ member), 890
usb_speed_t::value (C++ enum), 889
usb_speed_t::USB_SPEED_FULL (C++ enumerator), 889
usb_speed_t::USB_SPEED_LOW (C++ enumerator), 889
usb_standard_desc_t::bLength (C++ member), 891
usb_standard_desc_t::bLength (C++ member), 891
usb_standard_desc_t::bLength (C++ member), 891
usb_standard_desc_t::bLength (C++ member), 891
usb_standard_desc_t::val (C++ member), 891
usb_standard_desc_t::val (C++ member), 891
USB_STR_DESC_SIZE (C macro), 901
usb_str_desc_t::bDescriptorType (C++ member), 895
usb_str_desc_t::bLength (C++ member), 895
usb_str_desc_t::USB_DESC_ATTR (C++ member), 895
usb_str_desc_t::val (C++ member), 896
usb_str_desc_t::val (C++ member), 896
usb_str_desc_t::val (C++ member), 896
usb_subclass_vendor_spec_t (C macro), 900
usb_transfer_cb_t (C++ type), 888
USB_TRANSFER_FLAG_zero_pack (C macro), 887
usb_transfer_s (C++ struct), 886
 USB_TRANSFER_FLAG_zero_pack (C macro), 887
usb_transfer_s::actual_num_bytes (C++ member), 887
usb_transfer_s::bEndpointAddress (C++ member), 887
usb_transfer_s::callback (C++ member), 887
usb_transfer_s::context (C++ member), 887
usb_transfer_s::data_buffer (C++ member), 886
usb_transfer_s::data_buffer_size (C++ member), 886
usb_transfer_s::device_handle (C++ member), 887
usb_transfer_s::flags (C++ member), 887
usb_transfer_s::isoc_packet_desc (C++ member), 887
usb_transfer_s::num_bytes (C++ member), 886
usb_transfer_s::num_isoc_packets (C++ member), 887
usb_transfer_s::status (C++ member), 887
usb_transfer_s::timeout_ms (C++ member), 887
usb_transfer_status_t::value (C++ enum), 889
usb_transfer_status_t::USB_TRANSFER_STATUS_CANCELLED (C++ enumerator), 889
usb_transfer_status_t::USB_TRANSFER_STATUS_COMPLETED (C++ enumerator), 889
usb_transfer_status_t::USB_TRANSFER_STATUS_ERROR (C++ enumerator), 889
usb_transfer_status_t::USB_TRANSFER_STATUS_NO_DEVICE (C++ enumerator), 890
usb_transfer_status_t::USB_TRANSFER_STATUS_OVERFLOW (C++ enumerator), 889
usb_transfer_status_t::USB_TRANSFER_STATUS_SKIPPED (C++ enumerator), 890
usb_transfer_status_t::USB_TRANSFER_STATUS_STALLED (C++ enumerator), 890
usb_transfer_status_t::USB_TRANSFER_STATUS_TIMED_OUT (C++ enumerator), 889
usb_transfer_t (C++ type), 888
usb_transfer_t::USB_TRANSFER_TYPE_BULK (C++ enumerator), 889
usb_transfer_t::USB_TRANSFER_TYPE_CTRL (C++ enumerator), 889
usb_transfer_t::USB_TRANSFER_TYPE_INTR (C++ enumerator), 889
usb_transfer_t::USB_TRANSFER_TYPE_ISOCHRONOUS (C++ enumerator), 889
USB_W_VALUE_DT_CONFIG (C macro), 908
USB_W_VALUE_DT_Device (C macro), 908
USB_W_VALUE_DT_Device_Qualifier (C macro), 908
USB_W_VALUE_DT_ENDPOINT (C macro), 908
USB_W_VALUE_DT_INTERFACE (C macro), 908
USB_W_VALUE_DT_INTERFACE_power (C macro), 908
Index

USB_W_VALUE_DT_OTHER_SPEED_CONFIG (C macro), 898
USB_W_VALUE_DT_STRING (C macro), 898
uxQueueMessagesWaiting (C++ function), 1480
uxQueueMessagesWaitingFromISR (C++ function), 1483
uxQueueSpacesAvailable (C++ function), 1480
uxSemaphoreGetCount (C macro), 1509
uxSemaphoreGetCountFromISR (C macro), 1509
uxTaskGetNumberOfTasks (C++ function), 1458
uxTaskGetStackHighWaterMark (C++ function), 1459
uxTaskGetStackHighWaterMark2 (C++ function), 1459
uxTaskGetSystemState (C++ function), 1460
uxTaskPriorityGet (C++ function), 1453
uxTaskPriorityGetFromISR (C++ function), 1453
uxTimerGetReloadMode (C++ function), 1517

V
vApplicationGetIdleTaskMemory (C++ function), 1460
vApplicationGetTimerTaskMemory (C++ function), 1518
vendor_ie_data_t::element_id (C++ member), 302
vendor_ie_data_t::length (C++ member), 302
vendor_ie_data_t::payload (C++ member), 302
vendor_ie_data_t::vendor_oui (C++ member), 303
vendor_ie_data_t::vendor_oui_type (C++ member), 303
evEventGroupDelete (C++ function), 1534
evEventGroupDeleteWithCaps (C++ function), 1581
vMessageBufferDelete (C macro), 1553
vMessageBufferDeleteWithCaps (C++ function), 1581
vprintf_like_t (C++ type), 1644
vQueueAddToRegistry (C++ function), 1483
vQueueDelete (C++ function), 1480
vQueueDeleteWithCaps (C++ function), 1579
vQueueUnregisterQueue (C++ function), 1483
vRingbufferDelete (C++ function), 1571
vRingbufferDeleteWithCaps (C++ function), 1573
vRingbufferGetInfo (C++ function), 1572
vRingbufferReturnItem (C++ function), 1570
vRingbufferReturnItemFromISR (C++ function), 1570
vSemaphoreCreateBinary (C macro), 1496
vSemaphoreDelete (C macro), 1509
vSemaphoreDeleteWithCaps (C++ function), 1580
vStreamBufferDelete (C++ function), 1541
vStreamBufferDeleteWithCaps (C++ function), 1581
taskAllocateMPURegions (C++ function), 1450
taskDelay (C++ function), 1451
taskDelete (C++ function), 1450
taskDeleteWithCaps (C++ function), 1579
taskGenericNotifyGiveFromISR (C++ function), 1464
taskGetInfo (C++ function), 1454
taskGetRunTimeStats (C++ function), 1462
taskList (C++ function), 1462
taskNotifyGiveFromISR (C macro), 1473
taskNotifyGiveIndexedFromISR (C macro), 1473
taskPrioritySet (C++ function), 1454
taskResume (C++ function), 1456
taskSetApplicationTaskTag (C++ function), 1459
taskSetThreadLocalStoragePointer (C++ function), 1459
taskSetThreadLocalStoragePointerAndDelCallback (C++ function), 1578
taskSetTimeOutState (C++ function), 1466
taskSuspend (C++ function), 1455
taskSuspendAll (C++ function), 1457
timerSetReloadMode (C++ function), 1517
timerSetTimerID (C++ function), 1515

W
wifi_action_rx_cb_t (C++ type), 321
wifi_action_tx_req_t (C++ struct), 307
wifi_action_tx_req_t::data (C++ member), 308
wifi_action_tx_req_t::data_len (C++ member), 308
wifi_action_tx_req_t::dest_mac (C++ member), 308
wifi_action_tx_req_t::ifx (C++ member), 308
wifi_action_tx_req_t::no_ack (C++ member), 308
wifi_action_tx_req_t::rx_cb (C++ member), 308
wifi_active_scan_time_t (C++ struct), 295
wifi_active_scan_time_t::max (C++ member), 295
wifi_active_scan_time_t::min (C++ member), 295
WIFI_AMPDU_TX_ENABLED (C macro), 292
WIFI_APSDU_TX_ENABLED (C macro), 292
WIFI_AMSDU_TX_ENABLED (C macro), 292
wifant_config_t (C++ struct), 307
wifiant_config_t::enabled_ant0 (C++ member), 307
wifiant_config_t::enabled_ant1 (C++ member), 307
Index

| wifi_ant_config_t::rx_ant_default (C++ member), 307 |
| wifi_ant_config_t::rx_ant_mode (C++ member), 307 |
| wifi_ant_config_t::tx_ant_mode (C++ member), 307 |
| wifi_ant_gpio_config_t (C++ struct), 307 |
| wifi_ant_gpio_config_t::gpio_cfg (C++ member), 307 |
| wifi_ant_gpio_t (C++ struct), 307 |
| wifi_ant_gpio_t::gpio_num (C++ member), 307 |
| wifi_ap_record_t::authmode (C++ member), 296 |
| wifi_ap_record_t::bssid (C++ member), 296 |
| wifi_ap_record_t::country (C++ member), 297 |
| wifi_ap_record_t::ftm_initiator (C++ member), 297 |
| wifi_ap_record_t::ftm_responder (C++ member), 297 |
| wifi_ap_record_t::group_cipher (C++ member), 297 |
| wifi_ap_record_t::he_ap (C++ member), 297 |
| wifi_ap_record_t::pairwise_cipher (C++ member), 297 |
| wifi_ap_record_t::phy_11ax (C++ member), 297 |
| wifi_ap_record_t::phy_11b (C++ member), 297 |
| wifi_ap_record_t::phy_11g (C++ member), 297 |
| wifi_ap_record_t::phy_11n (C++ member), 297 |
| wifi_ap_record_t::phy_lr (C++ member), 297 |
| wifi_ap_record_t::primary (C++ member), 296 |
| wifi_ap_record_t::reserved (C++ member), 297 |
| wifi_ap_record_t::rssi (C++ member), 296 |
| wifi_ap_record_t::second (C++ member), 296 |
| wifi_ap_record_t::ssid (C++ member), 296 |
| wifi_ap_record_t::wps (C++ member), 297 |
| wifi_auth_mode_t::WIFI_AUTH_MAX (C++ member), 322 |
| wifi_auth_mode_t::WIFI_AUTH_ENTERPRISE (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_MAX (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_OPEN (C++ member), 322 |
| wifi_auth_mode_t::WIFI_AUTH_OWE (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WAPI_PSK (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WPA_PSK (C++ member), 322 |
| wifi_auth_mode_t::WIFI_AUTH_WPA2_ENTERPRISE (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WPA2_PSK (C++ member), 322 |
| wifi_auth_mode_t::WIFI_AUTH_WPA2_WPA3_PSK (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WPA2_WPA3_ENT_192 (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WPA3_EXT_PSK (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WPA3_PSK (C++ member), 323 |
| wifi_auth_mode_t::WIFI_AUTH_WPA_PSK (C++ member), 322 |

Espressif Systems

Submit Document Feedback

2311
Release master
Index

wifi_event_sta_authmode_change_t (C++ struct), 312
wifi_event_sta_authmode_change_t::new_mode (C++ member), 312
wifi_event_sta_authmode_change_t::old_mode (C++ member), 312
wifi_event_sta_connected_t (C++ struct), 311
wifi_event_sta_connected_t::aid (C++ member), 312
wifi_event_sta_connected_t::authmode (C++ member), 311
wifi_event_sta_connected_t::bssid (C++ member), 312
wifi_event_sta_connected_t::channel (C++ member), 312
wifi_event_sta_connected_t::ssid (C++ member), 311
wifi_event_sta_connected_t::ssid_len (C++ member), 311
wifi_event_sta_disconnected_t (C++ struct), 312
wifi_event_sta_disconnected_t::bssid (C++ member), 312
wifi_event_sta_disconnected_t::reason (C++ member), 312
wifi_event_sta_disconnected_t::rssi (C++ member), 312
wifi_event_sta_disconnected_t::ssid (C++ member), 312
wifi_event_sta_disconnected_t::ssid_len (C++ member), 312
wifi_event_sta_scan_done_t (C++ struct), 311
wifi_event_sta_scan_done_t::number (C++ member), 311
wifi_event_sta_scan_done_t::scan_id (C++ member), 311
wifi_event_sta_scan_done_t::status (C++ member), 311
wifi_event_sta_wps_er_pin_t (C++ struct), 312
wifi_event_sta_wps_er_pin_t::pin_code (C++ member), 313
wifi_event_sta_wps_er_success_t (C++ struct), 313
wifi_event_sta_wps_er_success_t::ap_cred (C++ member), 313
wifi_event_sta_wps_er_success_t::ap_cred_cnt (C++ member), 313
wifi_event_sta_wps_er_success_t::passphrase (C++ member), 313
wifi_event_sta_wps_er_success_t::ssid (C++ member), 313
wifi_event_sta_wps_fail_reason_t (C++ enum), 336
wifi_event_sta_wps_fail_reason_t::WPS_FAIL_REASON_RECV_M2D (C++ enumerator), 336
wifi_event_sta_wps_fail_reason_t::WPS_FAIL_REASON_NORMAL (C++ enumerator), 336
wifi_event_sta_wps_fail_reason_t::WPS_FAIL_REASON_MAX (C++ enumerator), 336
wifi_event_t::WIFI_EVENT_ACTION_TX_STATUS (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_AP_PROBEREQRECVED (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_AP_STACONNECTED (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_AP_STADISCONNECTED (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_AP_START (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_AP_STOP (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_AP_WPS_RG_FAILED (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_AP_WPS_RG_PBC_OVERLAP (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_AP_WPS_RG_PIN (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_AP_WPS_RG_SUCCESS (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_AP_WPS_RG_TIMEOUT (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_CONNECTIONLESS_MODULE_WAKE_INTERVAL_START (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_FTM_REPORT (C++ enumerator), 334
wifi_event_t::WIFI_EVENT_ITWT_PROBE (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_ITWT_SUSPEND (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_ITWT_TEARDOWN (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_ITWT_SETUP (C++ enumerator), 335
wifi_event_t::WIFI_EVENT_MAX (C++ enumerator), 336
wifi_event_t::WIFI_EVENT_NAN_LOGIN (C++ member), 333
wifi_event_t::WIFI_EVENT_NAN_MAX (C++ member), 333
wifi_event_t::WIFI_EVENT_NAN_SVC_MATCH (C++ member), 333
wifi_event_t::WIFI_EVENT_NAN_STOPPED (C++ member), 333
wifi_event_t::WIFI_EVENT_NAN_TIMER (C++ member), 333
wifi_event_t::WIFI_EVENT_NAN_TIMER_MAX (C++ member), 333
wifi_event_t::WIFI_EVENT_NDP_CONFIRM (C++ member), 335
wifi_event_t::WIFI_EVENT_NDP_CONFIRM_MAX (C++ member), 335
wifi_event_t::WIFI_EVENT_NDP_INDICATION (C++ member), 335
wifi_event_t::WIFI_EVENT_NDP_TERMINATED (C++ member), 335
wifi_event_t::WIFI_EVENT_NDP_TERMINATED_MAX (C++ member), 335

wifi_event_t::WIFI_EVENT_ROC_DONE
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_SCAN_DONE
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_AUTHMODE_CHANGE
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_BEACON_TIMEOUT
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_CHANGED
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_CONNECTED
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_DISCONNECTED
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_STOP
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_START
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_BEACON_TIMEOUT
(C++ enumerator), 334
wifi_event_t::WIFI_EVENT_STA_DISCONNECTED
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_DISCONNECTED
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_STOP
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_STOP
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296
wifi_event_t::WIFI_EVENT_STA_BSS_RSSI_LOW
(C++ member), 296

(C++ member), 289
wifi_init_config_t::static_tx_buf_num (C++ member), 289
wifi_init_config_t::tx_buf_type (C++ member), 289
wifi_init_config_t::wifi_task_core_id (C++ member), 290
wifi_init_config_t::wpa_crypto_funcs (C++ member), 289
wifi_interface_t (C++ enum), 322
wifi_interface_t::WIFI_IF_AP (C++ enumerator), 322
wifi_interface_t::WIFI_IF_MAX (C++ enumerator), 322
wifi_interface_t::WIFI_IF_STA (C++ enumerator), 322
WIFI_MGMT_SBUF_NUM (C macro), 293
wifi_mode_t (C++ enum), 321
wifi_mode_t::WIFI_MODE_AP (C++ enumerator), 322
wifi_mode_t::WIFI_MODE_APSTA (C++ enumerator), 322
wifi_mode_t::WIFI_MODE_MAX (C++ enumerator), 322
wifi_mode_t::WIFI_MODE_NAN (C++ enumerator), 322
wifi_mode_t::WIFI_MODE_NULL (C++ enumerator), 321
wifi_mode_t::WIFI_MODE_STA (C++ enumerator), 322
WIFI_NAN_CONFIG_DEFAULT (C macro), 355
wifi_nan_config_t (C++ struct), 301
wifi_nan_config_t::master_pref (C++ member), 301
wifi_nan_config_t::op_channel (C++ member), 301
wifi_nan_config_t::scan_time (C++ member), 301
wifi_nan_config_t::warm_up_sec (C++ member), 301
wifi_nan_datapath_end_req_t (C++ struct), 311
wifi_nan_datapath_end_req_t::ndp_id (C++ member), 311
wifi_nan_datapath_end_req_t::peer_mac (C++ member), 311
wifi_nan_followup_params_t (C++ struct), 310
wifi_nan_followup_params_t::inst_id (C++ member), 310
wifi_nan_followup_params_t::peer_id (C++ member), 310
wifi_nan_followup_params_t::peer_mac (C++ member), 310
wifi_nan_followup_params_t::svc_info (C++ member), 310
wifi_nan_publish_cfg_t (C++ struct), 309
wifi_nan_publish_cfg_t::datapath_request (C++ member), 309
wifi_nan_publish_cfg_t::matching_filter (C++ member), 309
wifi_nan_publish_cfg_t::reserved (C++ member), 309
wifi_nan_publish_cfg_t::service_name (C++ member), 309
wifi_nan_publish_cfg_t::single_replied_event (C++ member), 309
wifi_nan_publish_cfg_t::svc_info (C++ member), 309
wifi_nan_publish_cfg_t::type (C++ member), 309
wifi_nan_service_type_t (C++ enum), 330
wifi_nan_service_type_t::NAN_PUBLISH_SOLICITED (C++ enumerator), 331
wifi_nan_service_type_t::NAN_PUBLISH_UNSOLICITED (C++ enumerator), 331
wifi_nan_service_type_t::NAN_SUBSCRIBE_ACTIVE (C++ enumerator), 331
wifi_nan_service_type_t::NAN_SUBSCRIBE_PASSIVE (C++ enumerator), 331
wifi_nan_subscribe_cfg_t (C++ struct), 309
wifi_nan_subscribe_cfg_t::matching_filter (C++ member), 309
wifi_nan_subscribe_cfg_t::reserved (C++ member), 310
wifi_nan_subscribe_cfg_t::service_name (C++ member), 309
wifi_nan_subscribe_cfg_t::svc_info (C++ member), 310
wifi_nan_subscribe_cfg_t::type (C++ member), 309
WIFI_NANO_FORMAT_ENABLED (C macro), 292
WIFI_NVSM_ENABLED (C macro), 292
WIFI_OFFCHAN_TX_CANCEL (C macro), 319
WIFI_OFFCHAN_TX_REQ (C macro), 319
wifi_phy_mode_t (C++ enum), 329
wifi_phy_mode_t::WIFI_PHY_MODE_11B (C++ enumerator), 329
wifi_phy_mode_t::WIFI_PHY_MODE_11G (C++ enumerator), 330
wifi_phy_mode_t::WIFI_PHY_MODE_HE20 (C++ enumerator), 330
wifi_phy_mode_t::WIFI_PHY_MODE_HT20 (C++ enumerator), 330
wifi_phy_mode_t::WIFI_PHY_MODE_HT40 (C++ enumerator), 330
wifi_phy_mode_t::WIFI_PHY_MODE_LR (C++ enumerator), 329
wifi_phy_rate_t::WIFI_PHY_RATE_LORA_500K (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_LORA_250K (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_9M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_6M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_5M_S (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_5M_L (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_1M_L (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_11M_L (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_11M_S (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_12M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_18M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_1M_L (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_24M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_2M_L (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_2M_S (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_36M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_48M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_54M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_5M_L (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_5M_S (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_6M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_9M (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_LORA_250K (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_LORA_500K (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MAX (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS0_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS0_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS1_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS1_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS2_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS2_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS3_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS3_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS4_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS4_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS5_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS5_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS6_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS6_SGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS7_LGI (C++ enumerator), 328
wifi_phy_rate_t::WIFI_PHY_RATE_MCS7_SGI (C++ enumerator), 328
wifi_pkt_rx_ctrl_t::not_sounding (C++ member), 303
wifi_pkt_rx_ctrl_t::__pad0__ (C++ member), 303
wifi_pkt_rx_ctrl_t::__pad10__ (C++ member), 305
wifi_pkt_rx_ctrl_t::__pad1__ (C++ member), 303
wifi_pkt_rx_ctrl_t::__pad2__ (C++ member), 303
wifi_pkt_rx_ctrl_t::__pad3__ (C++ member), 304
wifi_pkt_rx_ctrl_t::__pad4__ (C++ member), 304
wifi_pkt_rx_ctrl_t::__pad5__ (C++ member), 304
wifi_pkt_rx_ctrl_t::__pad6__ (C++ member), 304
wifi_pkt_rx_ctrl_t::__pad7__ (C++ member), 304
wifi_pkt_rx_ctrl_t::__pad8__ (C++ member), 304
wifi_pkt_rx_ctrl_t::__pad9__ (C++ member), 305
wifi_pkt_rx_ctrl_t::aggregation (C++ member), 304
wifi_pkt_rx_ctrl_t::ampdu_cnt (C++ member), 304
wifi_pkt_rx_ctrl_t::ant (C++ member), 304
wifi_pkt_rx_ctrl_t::channel (C++ member), 304
wifi_pkt_rx_ctrl_t::cwb (C++ member), 303
wifi_pkt_rx_ctrl_t::fec_coding (C++ member), 304
wifi_pkt_rx_ctrl_t::mcs (C++ member), 303
wifi_pkt_rx_ctrl_t::noise_floor (C++ member), 304
wifi_pkt_rx_ctrl_t::not_sounding (C++ member), 304
member), 303
wifi_pkt_rx_ctrl_t::rate (C++ member), 303
wifi_pkt_rx_ctrl_t::rssi (C++ member), 303
wifi_pkt_rx_ctrl_t::rx_state (C++ member), 305
wifi_pkt_rx_ctrl_t::secondary_channel (C++ member), 304
wifi_pkt_rx_ctrl_t::sgi (C++ member), 304
wifi_pkt_rx_ctrl_t::sig_len (C++ member), 305
wifi_pkt_rx_ctrl_t::sig_mode (C++ member), 303
wifi_pkt_rx_ctrl_t::smoothing (C++ member), 304
wifi_pkt_rx_ctrl_t::stbc (C++ member), 304
wifi_pkt_rx_ctrl_t::timestamp (C++ member), 304
wifi_pmfs_config_t (C++ struct), 298
wifi_pmfs_config_t::capable (C++ member), 298
wifi_pmfs_config_t::required (C++ member), 298
WIFI_PROMIS_CTRL_FILTER_MASK_ACK (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_ALL (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_BAR (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_CE (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_CFEND (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_CFENDACK (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_CTS (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_PSPOLL (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_RTS (C macro), 320
WIFI_PROMIS_CTRL_FILTER_MASK_WRAPPER (C macro), 320
WIFI_PROMIS_FILTER_MASK_ALL (C macro), 319
WIFI_PROMIS_FILTER_MASK_CTRL (C macro), 319
WIFI_PROMIS_FILTER_MASK_DATA (C macro), 319
WIFI_PROMIS_FILTER_MASK_DATA_AMPDU (C macro), 320
WIFI_PROMIS_FILTER_MASK_DATA_MPU (C macro), 319
WIFI_PROMIS_FILTER_MASK_FCSFAIL (C macro), 320
WIFI_PROMIS_FILTER_MASK_MGMT (C macro), 319
wfi_promiscuous_cb_t (C++ type), 293
wfi_promiscuous_ctrl_t (C++ struct), 305
wfi_promiscuous_filter_t::filter_mask (C++ member), 305
wfi_promiscuous_filter_t::payload (C++ member), 305
wfi_promiscuous_pkt_t (C++ struct), 305
wfi_promiscuous_pkt_t::payload (C++ member), 305
wfi_promiscuous_pkt_type_t (C++ enum), 330
wfi_promiscuous_pkt_type_t::WIFI_PKT_ACCESS (C++ enumerator), 330
wfi_promiscuous_pkt_type_t::WIFI_PKT_DATA (C++ enumerator), 330
wfi_promiscuous_pkt_type_t::WIFI_PKT_MGMT (C++ enumerator), 330
wfi_promiscuous_pkt_type_t::WIFI_PKT_MISC (C++ enumerator), 330
WIFI_PROTOCOL_11A (C macro), 319
WIFI_PROTOCOL_11B (C macro), 319
WIFI_PROTOCOL_11G (C macro), 319
WIFI_PROTOCOL_11N (C macro), 319
WIFI_PROTOCOL_LR (C macro), 319
wifi_prov_config_data_handler (C++ function), 1263
wifi_prov_config_get_data_t (C++ struct), 1263
wifi_prov_config_get_data_t::conn_info (C++ member), 1263
wifi_prov_config_get_data_t::fail_reason (C++ member), 1263
wifi_prov_config_get_data_t::wifi_state (C++ member), 1263
wifi_prov_config_handlers (C++ struct), 1264
wifi_prov_config_handlers::apply_config_handler (C++ member), 1264
wifi_prov_config_handlers::ctx (C++ member), 1264

Espressif Systems

Submit Document Feedback

Release master
# Index

- **WIFI_STATIS_ALL** (C macro), 321  
- **WIFI_STATIS_BUFFER** (C macro), 321  
- **WIFI_STATIS_DIAG** (C macro), 321  
- **WIFI_STATIS_PS** (C macro), 321  
- **WIFI_STATIC_TX_BUFFER_NUM** (C macro), 292  
- **wifi_storage_t::WIFI_STORAGE_FLASH** (C++ enumerator), 329  
- **wifi_storage_t::WIFI_STORAGE_RAM** (C++ enumerator), 329  
- **wifi_storage_t::WIFI_STORAGE_RAM_MEMBER** (C++ function), 329  
- **wifi_storage_t::WIFI_STORAGE_FLASH_MEMBER** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_BSS戕** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_ASSOC** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_PROBE** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_BEACON** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_DISCONN** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE лид** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_FOUR** (C++ function), 329  
- **wifi_vendor_type_t::WIFI_VENDOR_IE_TYPE_THREE** (C++ function), 329  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_0** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_1** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_2** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_3** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_4** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_5** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_6** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_7** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_8** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_9** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_10** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_11** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_12** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_13** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_14** (C++ macro), 299  
- **wifivendor_ie_id_t::WIFI_VND_IE_ID_15** (C++ macro), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_BSS戕** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_ASSOC** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_PROBE** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_BEACON** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_DISCONN** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE лид** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_FOUR** (C++ function), 299  
- **wifivendor_ie_type_t::WIFI_VND_IE_TYPE_THREE** (C++ function), 299  
- **wl_erase_range(C++ function), 1354**  
- **wl_handle_t(C++ type), 1355**  
- **WL_INVALID_HANDLE(C macro), 1355**  
- **wl_mount(C++ function), 1353**  
- **wl_read(C++ function), 1354**  
- **wl_sector_size(C++ function), 1355**  
- **wl_size(C++ function), 1354**  
- **wl_unmount(C++ function), 1353**  
- **wl_write(C++ function), 1354**  
- **WPS_CONFIG_INIT_DEFAULT(C macro), 444**  
- **wps_factory_information_t::device_name(C++ member), 343**  
- **wps_factory_information_t::manufacturer(C++ member), 343**  
- **wps_factory_information_t::model_name(C++ member), 343**  
- **wps_factory_information_t::model_number(C++ member), 343**  
- **wps_factory_information_t::WPS_MAX_DEVICE_NAME_LEN(C macro), 344**  
- **wps_factory_information_t::WPS_MAX_MANUFACTURER_LEN(C macro), 344**  

---

Espressif Systems 2322 
Submit Document Feedback
xStreamBufferSendCompletedFromISR (C++ function), 1537
xStreamBufferSend (C++ function), 1537
xRingbufferSend (C++ function), 1566
xRingbufferSendAcquire (C++ function), 1566
xRingbufferSendComplete (C++ function), 1567
xRingbufferSendFromISR (C++ function), 1566
xSemaphoreCreateBinary (C macro), 1496
xSemaphoreCreateBinaryStatic (C macro), 1497
xSemaphoreCreateBinaryWithCaps (C++ function), 1579
xSemaphoreCreateCounting (C macro), 1507
xSemaphoreCreateCountingStatic (C macro), 1508
xSemaphoreCreateCountingWithCaps (C++ function), 1579
xSemaphoreCreateMutex (C macro), 1504
xSemaphoreCreateMutexStatic (C macro), 1505
xSemaphoreCreateMutexWithCaps (C++ function), 1580
xSemaphoreCreateRecursiveMutex (C macro), 1505
xSemaphoreCreateRecursiveMutexStatic (C macro), 1506
xSemaphoreCreateRecursiveMutexWithCaps (C++ function), 1580
xSemaphoreGetMutexHolder (C macro), 1509
xSemaphoreGetMutexHolderFromISR (C macro), 1509
xSemaphoreGetStaticBuffer (C macro), 1510
xSemaphoreGive (C macro), 1500
xSemaphoreGiveFromISR (C macro), 1502
xSemaphoreGiveRecursive (C macro), 1501
xSemaphoreTake (C macro), 1498
xSemaphoreTakeFromISR (C macro), 1504
xSemaphoreTakeRecursive (C macro), 1499
xSTATIC_RINGBUFFER (C++ struct), 1573
xStreamBufferBytesAvailable (C++ function), 1542
xStreamBufferCreateStaticWithCallback (C macro), 1544
xStreamBufferCreateWithCallback (C macro), 1543
xStreamBufferCreateWithCaps (C++ function), 1580
xStreamBufferGetStaticBuffers (C++ function), 1537
xStreamBufferIsEmpty (C++ function), 1541
xStreamBufferIsFull (C++ function), 1541
xStreamBufferReceive (C++ function), 1539
xStreamBufferReceiveCompletedFromISR (C++ function), 1543
xStreamBufferReceiveFromISR (C++ function), 1540
xStreamBufferReset (C++ function), 1542
xStreamBufferSend (C++ function), 1537
xStreamBufferSendCompletedFromISR (C++ function), 1542
xStreamBufferSendFromISR (C++ function), 1538
xStreamBufferSetTriggerLevel (C++ function), 1542
xStreamBufferSpacesAvailable (C++ function), 1542
xTASK_STATUS (C++ struct), 1468
xTASK_STATUS::eCurrentTaskHandle (C++ member), 1458
xTASK_STATUS::pxStackName (C++ member), 1468
xTASK_STATUS::pxStackName (C++ member), 1469
xTASK_STATUS::ulRunTimeCounter (C++ member), 1469
xTASK_STATUS::usStackHighWaterMark (C++ member), 1469
xTASK_STATUS::uxCurrentPriority (C++ member), 1468
xTASK_STATUS::uxCurrentPriority (C++ member), 1468
xTASK_STATUS::xCoreID (C++ member), 1469
xTASK_STATUS::xHandle (C++ member), 1468
xTASK_STATUS::xTaskNumber (C++ member), 1468
xTaskAbortDelay (C++ function), 1452
xTaskCallApplicationTaskHook (C++ function), 1460
xTaskCheckForTimeOut (C++ function), 1467
xTaskCreate (C++ function), 1447
xTaskCreatePinnedToCore (C++ function), 1576
xTaskCreatePinnedToCoreWithCaps (C++ function), 1578
xTaskCreateStatic (C++ function), 1448
xTaskCreateStaticPinnedToCore (C++ function), 1576
xTaskCreateWithCaps (C++ function), 1578
xTaskDelayUntil (C++ function), 1452
xTaskGenericNotifyStateClear (C++ function), 1465
xTaskGenericNotifyWait (C++ function), 1463
xTaskGetApplicationTaskTag (C++ function), 1459
xTaskGetApplicationTaskTagFromISR (C++ function), 1459
xTaskGetCoreID (C++ function), 1459
xTaskGetCurrentTaskHandleForCore (C++ function), 1577
xTaskGetHandle (C++ function), 1458
xTaskGetIdleTaskHandle (C++ function), 1460
xTaskGetIdleTaskHandleForCore (C++ function), 1577
xTaskGetStaticBuffers (C++ function), 1459
xTaskGetTickCount (C++ function), 1458
xTaskGetTickCountFromISR (C++ function), 1459
xTaskNotifyAndQueryIndexed (C macro), 1471
xTaskNotifyAndQueryIndexedFromISR (C macro), 1472
xTaskNotifyGiveIndexed (C macro), 1473
xTaskNotifyIndexed (C macro), 1470
xTaskNotifyIndexedFromISR (C macro), 1471
xTaskNotifyStateClear (C macro), 1475
xTaskNotifyStateClearIndexed (C macro), 1475
xTaskNotifyWait (C macro), 1473
xTaskNotifyWaitIndexed (C macro), 1473
xTaskResumeAll (C++ function), 1457
xTaskResumeFromISR (C++ function), 1456
xtensa_perfmon_config (C++ struct), 1676
xtensa_perfmon_config::call_function (C++ member), 1676
xtensa_perfmon_config::call_params (C++ member), 1676
xtensa_perfmon_config::callback (C++ member), 1676
xtensa_perfmon_config::callback_params (C++ member), 1676
xtensa_perfmon_config::counters_size (C++ member), 1676
xtensa_perfmon_config::max_deviation (C++ member), 1676
xtensa_perfmon_config::repeat_count (C++ member), 1676
xtensa_perfmon_config::select_mask (C++ member), 1676
xtensa_perfmon_config::tracelevel (C++ member), 1676
xtensa_perfmon_config_t (C++ type), 1677
xtensa_perfmon_dump (C++ function), 1675
xtensa_perfmon_exec (C++ function), 1675
xtensa_perfmon_init (C++ function), 1674
xtensa_perfmon_overflow (C++ function), 1675
xtensa_perfmon_reset (C++ function), 1674
xtensa_perfmon_start (C++ function), 1674
xtensa_perfmon_stop (C++ function), 1675
xtensa_perfmon_value (C++ function), 1675
xtensa_perfmon_view_cb (C++ function), 1675
xTimerChangePeriod (C macro), 1519
xTimerChangePeriodFromISR (C macro), 1525
xTimerCreate (C++ function), 1510
xTimerCreateStatic (C++ function), 1512
xTimerDelete (C macro), 1520
xTimerGetExpiryTime (C++ function), 1518
xTimerGetPeriod (C++ function), 1517
xTimerGetReloadMode (C++ function), 1517
xTimerGetStaticBuffer (C++ function), 1518
xTimerGetTimerDaemonTaskHandle (C++ function), 1515
xTimerIsTimerActive (C++ function), 1515
xTimerPendFunctionCall (C++ function), 1517