

---

# **Read the Docs Template Documentation**

*Release v3.0.8-30-gf3704f027*

**Read the Docs**

**Oct 30, 2019**



<b>1</b>	<b>Get Started</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	What You Need . . . . .	3
1.3	Guides . . . . .	4
1.4	Setup Toolchain . . . . .	32
1.5	Get ESP-IDF . . . . .	41
1.6	Setup Path to ESP-IDF . . . . .	42
1.7	Start a Project . . . . .	42
1.8	Connect . . . . .	42
1.9	Configure . . . . .	42
1.10	Build and Flash . . . . .	43
1.11	Monitor . . . . .	44
1.12	Updating ESP-IDF . . . . .	45
1.13	Related Documents . . . . .	45
<b>2</b>	<b>API Reference</b>	<b>61</b>
2.1	Wi-Fi API . . . . .	61
2.2	Bluetooth API . . . . .	98
2.3	Ethernet API . . . . .	193
2.4	Peripherals API . . . . .	201
2.5	Protocols API . . . . .	391
2.6	Storage API . . . . .	402
2.7	System API . . . . .	449
2.8	Configuration Options . . . . .	610
<b>3</b>	<b>ESP32 Hardware Reference</b>	<b>685</b>
3.1	ESP32 Modules and Boards . . . . .	685
3.2	Previous Versions of ESP32 Modules and Boards . . . . .	689
<b>4</b>	<b>API Guides</b>	<b>697</b>
4.1	General Notes About ESP-IDF Programming . . . . .	697
4.2	Build System . . . . .	700
4.3	Deep Sleep Wake Stubs . . . . .	713
4.4	ESP32 Core Dump . . . . .	714
4.5	Flash Encryption . . . . .	716
4.6	ESP-IDF FreeRTOS SMP Changes . . . . .	725
4.7	High-Level Interrupts . . . . .	732

4.8	JTAG Debugging . . . . .	733
4.9	Partition Tables . . . . .	789
4.10	Secure Boot . . . . .	793
4.11	ULP coprocessor programming . . . . .	798
4.12	Unit Testing in ESP32 . . . . .	822
4.13	Console . . . . .	823
4.14	ESP32 ROM console . . . . .	826
4.15	Wi-Fi Driver . . . . .	829
4.16	Support for external RAM . . . . .	859
<b>5</b>	<b>Contributions Guide</b>	<b>863</b>
5.1	How to Contribute . . . . .	863
5.2	Before Contributing . . . . .	863
5.3	Pull Request Process . . . . .	864
5.4	Legal Part . . . . .	864
5.5	Related Documents . . . . .	864
<b>6</b>	<b>ESP-IDF Versions</b>	<b>881</b>
6.1	Releases . . . . .	881
6.2	Which Version Should I Start With? . . . . .	882
6.3	Versioning Scheme . . . . .	883
6.4	Checking The Current Version . . . . .	883
6.5	Git Workflow . . . . .	883
6.6	Updating ESP-IDF . . . . .	884
<b>7</b>	<b>Resources</b>	<b>887</b>
<b>8</b>	<b>Copyrights and Licenses</b>	<b>889</b>
8.1	Software Copyrights . . . . .	889
8.2	ROM Source Code Copyrights . . . . .	890
8.3	Xtensa libhal MIT License . . . . .	890
8.4	TinyBasic Plus MIT License . . . . .	890
8.5	TJpgDec License . . . . .	891
<b>9</b>	<b>About</b>	<b>893</b>
	<b>Index</b>	<b>895</b>

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

<a href="#">Get Started</a>	<a href="#">API Reference</a>	<a href="#">H/W Reference</a>
<a href="#">API Guides</a>	<a href="#">Contribute</a>	<a href="#">Resources</a>



This document is intended to help users set up the software environment for development of applications using hardware based on the Espressif ESP32. Through a simple example we would like to illustrate how to use ESP-IDF (Espressif IoT Development Framework), including the menu based configuration, compiling the ESP-IDF and firmware download to ESP32 boards.

---

**Note:** This is documentation for branch `release/v3.0` of ESP-IDF. Other *ESP-IDF Versions* are also available.

---

## 1.1 Introduction

ESP32 integrates Wi-Fi (2.4 GHz band) and Bluetooth 4.2 solutions on a single chip, along with dual high performance cores, Ultra Low Power co-processor and several peripherals. Powered by 40 nm technology, ESP32 provides a robust, highly integrated platform to meet the continuous demands for efficient power usage, compact design, security, high performance, and reliability.

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

## 1.2 What You Need

To develop applications for ESP32 you need:

- **PC** loaded with either Windows, Linux or Mac operating system
- **Toolchain** to build the **Application** for ESP32
- **ESP-IDF** that essentially contains API for ESP32 and scripts to operate the **Toolchain**
- A text editor to write programs (**Projects**) in C, e.g. [Eclipse](#)

- The **ESP32** board itself and a **USB cable** to connect it to the **PC**

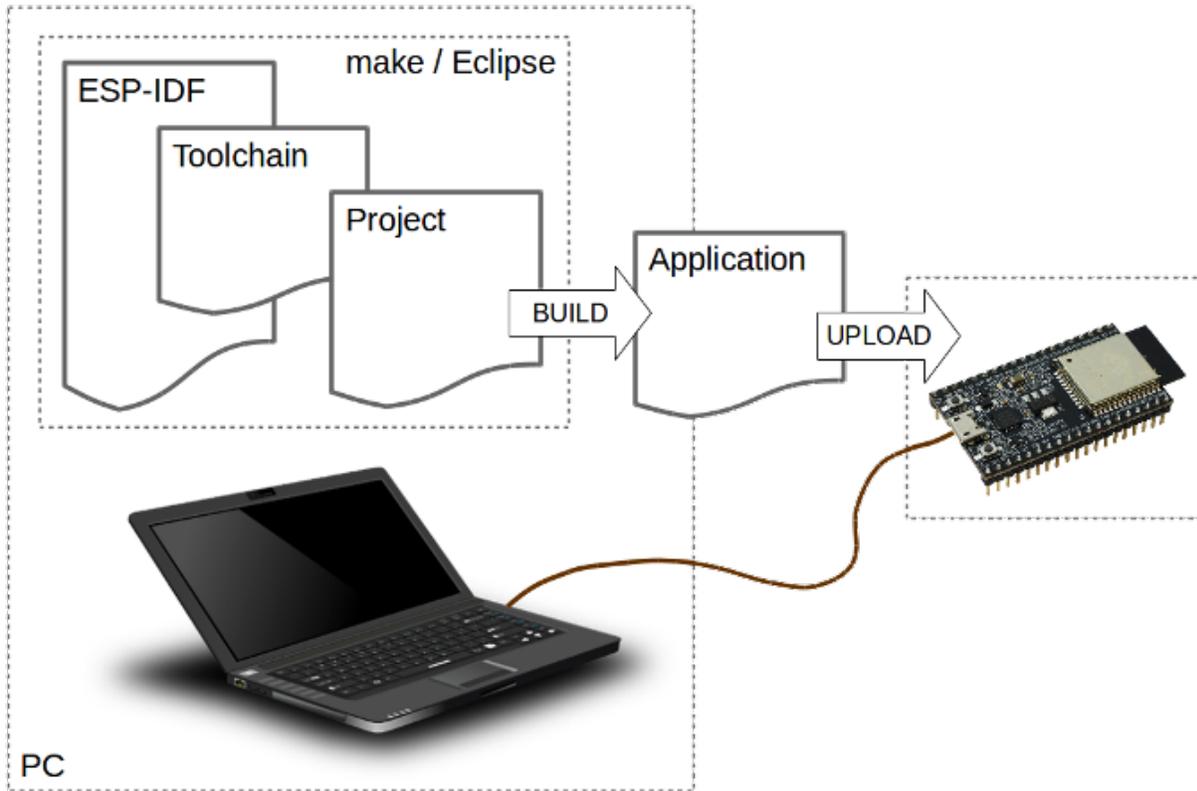


Fig. 1: Development of applications for ESP32

Preparation of development environment consists of three steps:

1. Setup of **Toolchain**
2. Getting of **ESP-IDF** from GitHub
3. Installation and configuration of **Eclipse**

You may skip the last step, if you prefer to use different editor.

Having environment set up, you are ready to start the most interesting part - the application development. This process may be summarized in four steps:

1. Configuration of a **Project** and writing the code
2. Compilation of the **Project** and linking it to build an **Application**
3. Flashing (uploading) of the **Application** to **ESP32**
4. Monitoring / debugging of the **Application**

See instructions below that will walk you through these steps.

## 1.3 Guides

If you have one of ESP32 development boards listed below, click on provided links to get you up and running.

## 1.3.1 ESP32-DevKitC Getting Started Guide

This user guide shows how to get started with ESP32-DevKitC development board.

### What You Need

- 1 × *ESP32-DevKitC board*
- 1 × USB A / micro USB B cable
- 1 × PC loaded with Windows, Linux or Mac OS

### Overview

ESP32-DevKitC is a small-sized ESP32-based development board produced by [Espressif](#). Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can connect these pins to peripherals as needed. Standard headers also make development easy and convenient when using a breadboard.

### Functional Description

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

**ESP-WROOM-32** Standard [ESP-WROOM-32](#) module soldered to the ESP32-DevKitC board.

**EN** Reset button: pressing this button resets the system.

**Boot** Download button: holding down the **Boot** button and pressing the **EN** button initiates the firmware download mode. Then user can download firmware through the serial port.

**USB** USB interface. It functions as the power supply for the board and the communication interface between PC and ESP-WROOM-32.

**I/O** Most of the pins on the ESP-WROOM-32 are broken out to the pin headers on the board. Users can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

### Power Supply Options

There following options are available to provide power supply to the ESP32-PICO-KIT V4:

1. Micro USB port, this is default power supply connection
2. 5V / GND header pins
3. 3V3 / GND header pins

**Warning:** Above options are mutually exclusive, i.e. the power supply may be provided using only one of the above options. Attempt to power the board using more than one connection at a time may damage the board and/or the power supply source.

### Start Application Development

Before powering up the ESP32-DevKitC, please make sure that the board has been received in good condition with no obvious signs of damage.

To start development of applications, proceed to section *Get Started*, that will walk you through the following steps:

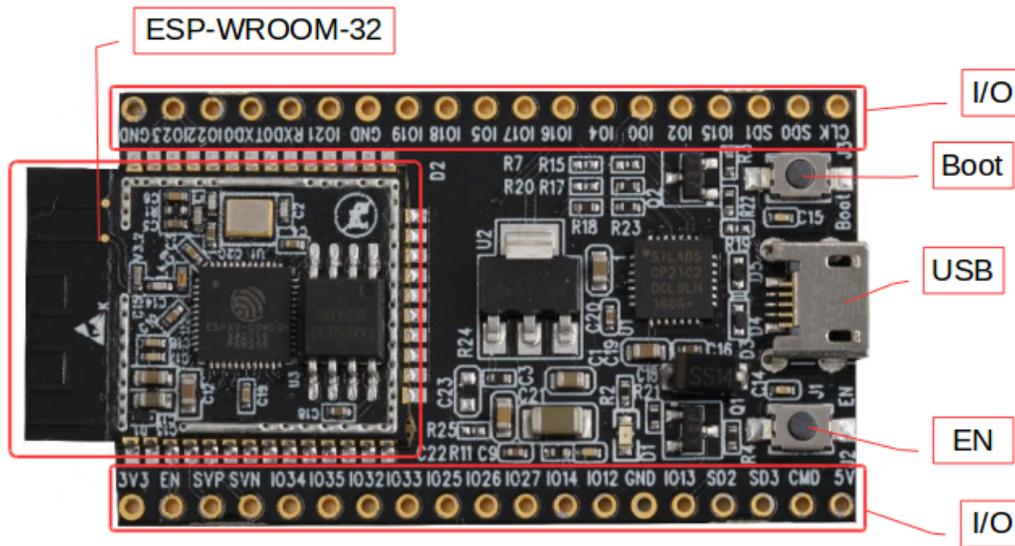


Fig. 2: ESP32-DevKitC board layout

- *Setup Toolchain* in your PC to develop applications for ESP32 in C language
- *Connect* the module to the PC and verify if it is accessible
- *Build and Flash* an example application to the ESP32
- *Monitor* instantly what the application is doing

## Related Documents

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

## 1.3.2 ESP-WROVER-KIT V3 Getting Started Guide

This user guide shows how to get started with ESP-WROVER-KIT V3 development board including description of its functionality and configuration options. For description of other versions of the ESP-WROVER-KIT check [ESP32 Hardware Reference](#).

If you like to start using this board right now, go directly to section [Start Application Development](#).

### What You Need

- 1 × *ESP-WROVER-KIT V3 board*
- 1 x Micro USB 2.0 Cable, Type A to Micro B

- 1 × PC loaded with Windows, Linux or Mac OS

## Overview

The ESP-WROVER-KIT is a development board produced by Espressif built around ESP32. This board is compatible with ESP32 modules, including the ESP-WROOM-32 and ESP32-WROVER. The ESP-WROVER-KIT features support for an LCD and MicroSD card. The I/O pins have been broken out from the ESP32 module for easy extension. The board carries an advanced multi-protocol USB bridge (the FTDI FT2232HL), enabling developers to use JTAG directly to debug the ESP32 through the USB interface. The development board makes secondary development easy and cost-effective.

**Note:** ESP-WROVER-KIT V3 integrates the ESP32-WROVER module by default.

## Functionality Overview

Block diagram below presents main components of ESP-WROVER-KIT and interconnections between components.

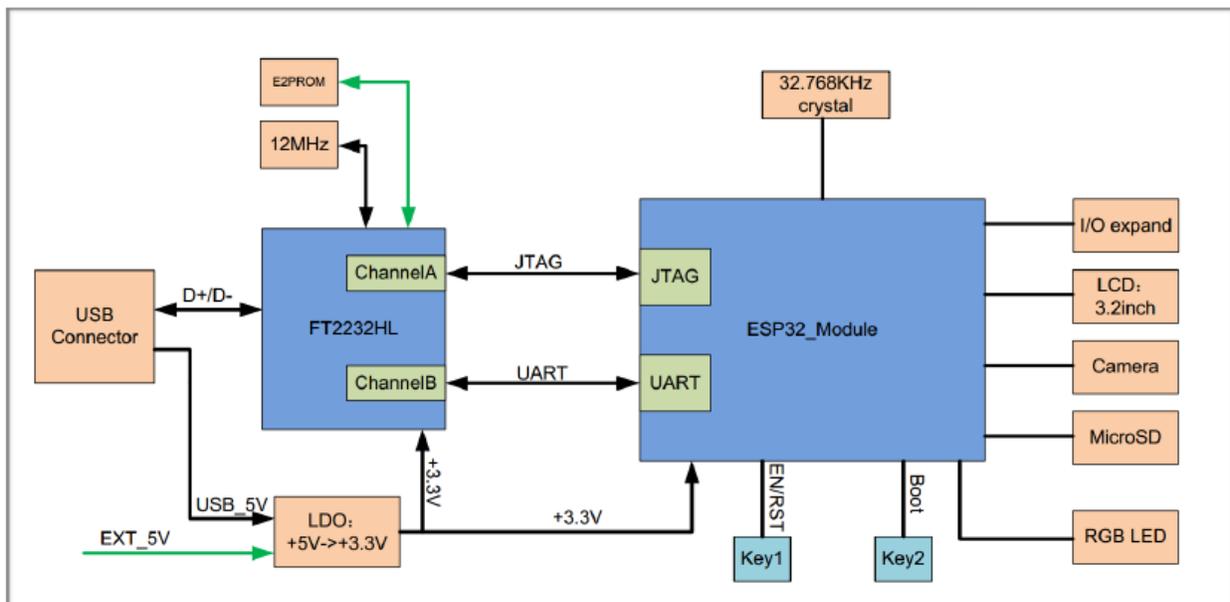


Fig. 3: ESP-WROVER-KIT block diagram

## Functional Description

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

**32.768 kHz** An external precision 32.768 kHz crystal oscillator provides the chip with a clock of low-power consumption during the Deep-sleep mode.

**0R** A zero Ohm resistor intended as a placeholder for a current shunt. May be desoldered or replaced with a current shunt to facilitate measurement of current required by ESP32 module depending on power mode.

**ESP32 Module** ESP-WROVER-KIT is compatible with both ESP-WROOM-32 and ESP32-WROVER. The ESP32-WROVER module features all the functions of ESP-WROOM-32 and integrates an external 32-MBit PSRAM for flexible extended storage and data processing capabilities.

---

**Note:** GPIO16 and GPIO17 are used as the CS and clock signal for PSRAM. To ensure reliable performance, the two GPIOs are not broken out.

---

**FT2232** The FT2232 chip is a multi-protocol USB-to-serial bridge. Users can control and program the FT2232 chip through the USB interface to establish communication with ESP32. The FT2232 chip also features USB-to-JTAG interface. USB-to-JTAG is available on channel A of FT2232, USB-to-serial on channel B. The embedded FT2232 chip is one of the distinguishing features of the ESPWROVER-KIT. It enhances users' convenience in terms of application development and debugging. In addition, users do not need to buy a JTAG debugger separately, which reduces the development cost, see [ESP-WROVER-KIT V3 schematic](#).

**UART** Serial port: the serial TX/RX signals on FT2232HL and ESP32 are broken out to the two sides of JP11. By default, the two signals are connected with jumpers. To use the ESP32 module serial interface only, the jumpers may be removed and the module can be connected to another external serial device.

**SPI** SPI interface: the SPI interface connects to an external flash (PSRAM). To interface another SPI device, an extra CS signal is needed. The electrical level on the flash of this module is 1.8V. If an ESP-WROOM-32 is being used, please note that the electrical level on the flash of this module is 3.3V.

**CTS/RTS** Serial port flow control signals: the pins are not connected to the circuitry by default. To enable them, respective pins of JP14 must be shorted with jumpers.

**JTAG** JTAG interface: the JTAG signals on FT2232HL and ESP32 are broken out to the two sides of JP8. By default, the two signals are disconnected. To enable JTAG, shorting jumpers are required on the signals.

**EN** Reset button: pressing this button resets the system.

**Boot** Download button: holding down the **Boot** button and pressing the **EN** button initiates the firmware download mode. Then user can download firmware through the serial port.

**USB** USB interface. It functions as the power supply for the board and the communication interface between PC and ESP32 module.

**Power Select** Power supply selection interface: the ESP-WROVER-KIT can be powered through the USB interface or the 5V Input interface. The user can select the power supply with a jumper. More details can be found in section [Setup Options](#), jumper header JP7.

**Power Key** Power on/off button: toggling to the right powers the board on; toggling to the left powers the board off.

**5V Input** The 5V power supply interface is used as a backup power supply in case of full-load operation.

**LDO** NCP1117(1A). 5V-to-3.3V LDO. (There is an alternative pin-compatible LDO — LM317DCY, with an output current of up to 1.5A). NCP1117 can provide a maximum current of 1A. The LDO solutions are available with both fixed output voltage and variable output voltage. For details please refer to [ESP-WROVER-KIT V3 schematic](#).

**Camera** Camera interface: a standard OV7670 camera module is supported.

**RGB** Red, green and blue (RGB) light emitting diodes (LEDs), which may be controlled by pulse width modulation (PWM).

**I/O** All the pins on the ESP32 module are led out to the pin headers on the ESP-WROVER-KIT. Users can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

**Micro SD Card** Micro SD card slot for data storage.

**LCD** ESP-WROVER-KIT supports mounting and interfacing a 3.2" SPI (standard 4-wire Serial Peripheral Interface) LCD, as shown on figure [ESP-WROVER-KIT board layout - back](#).

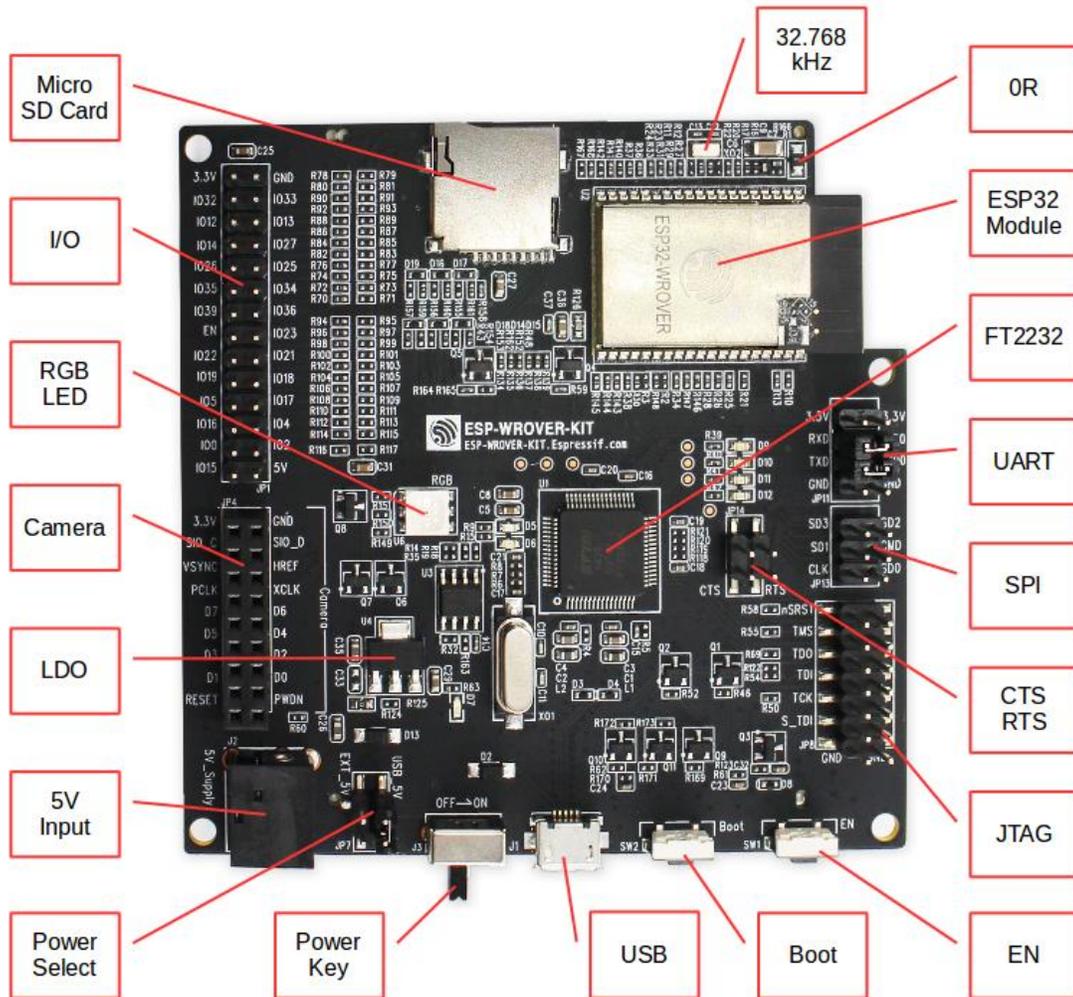


Fig. 4: ESP-WROVER-KIT board layout - front

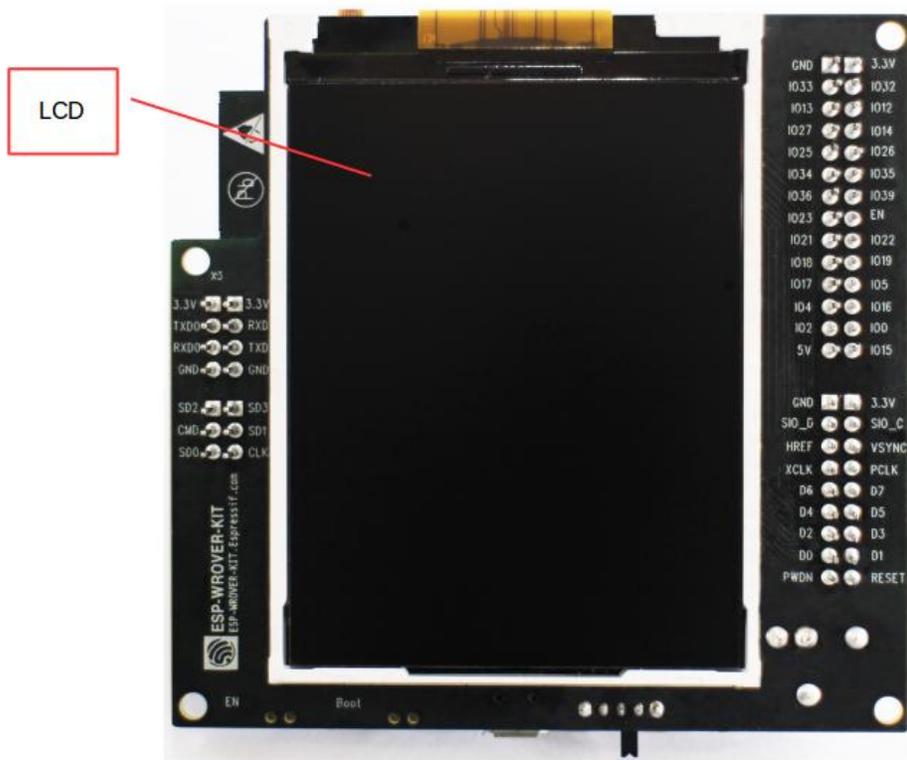
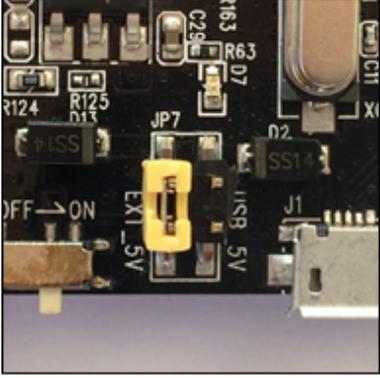
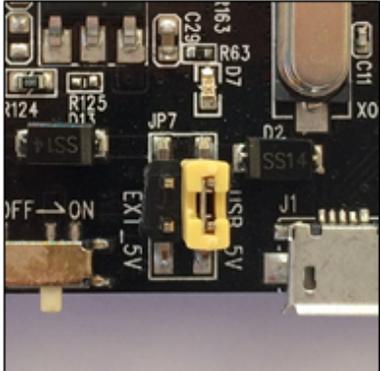
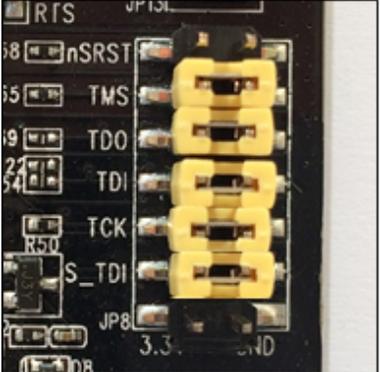
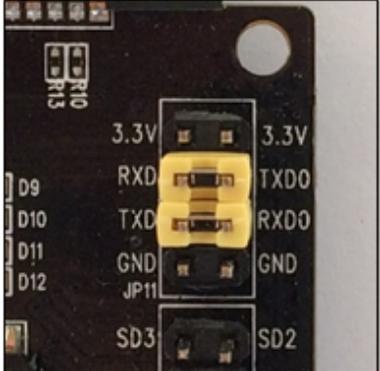
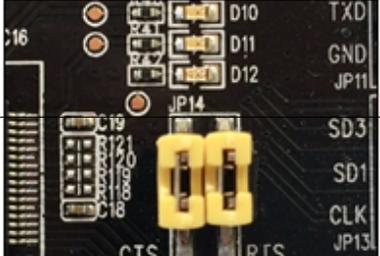


Fig. 5: ESP-WROVER-KIT board layout - back

## Setup Options

There are five jumper headers available to set up the board functionality. Typical options to select from are listed in table below.

Header	Jumper Setting	Description of Functionality
JP7		Power ESP-WROVER-KIT board from an external   power supply
JP7		Power ESP-WROVER-KIT board from an USB port
JP8		Enable JTAG functionality
JP11		Enable UART communication
12		Chapter 1. Get Started

## Allocation of ESP32 Pins

Several pins / terminals of ESP32 module are allocated to the on board hardware. Some of them, like GPIO0 or GPIO2, have multiple functions. If certain hardware is not installed, e.g. nothing is plugged in to the Camera / JP4 header, then selected GPIOs may be used for other purposes.

### Main I/O Connector / JP1

The JP1 connector is shown in two columns in the middle under “I/O” headers. The two columns “Shared With” outside, describe where else on the board certain GPIO is used.

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

Legend:

- NC/XTAL - *32.768 kHz Oscillator*
- JTAG - *JTAG / JP8*
- Boot - *Boot button / SW2*
- Camera - *Camera / JP4*
- LED - *RGB LED*
- MicroSD - *MicroSD Card / J4*
- LCD - *LCD / U5*
- PSRAM - *ESP32-WROVER’s PSRAM, if ESP32-WROVER is installed*

### 32.768 kHz Oscillator

	ESP32 Pin
1	GPIO32
2	GPIO33

**Note:** As GPIO32 and GPIO33 are connected to the oscillator, to maintain signal integrity, they are not connected to JP1 I/O expansion connector. This allocation may be changed from oscillator to JP1 by desoldering 0R resistors from positions R11 / R23 and installing them in positions R12 / R24.

---

### SPI Flash / JP13

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

### JTAG / JP8

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

### Camera / JP4

	ESP32 Pin	Camera Signal
1	GPIO27	SCCB Clock
2	GPIO26	SCCB Data
3	GPIO21	System Clock
4	GPIO25	Vertical Sync
5	GPIO23	Horizontal Reference
6	GPIO22	Pixel Clock
7	GPIO4	Pixel Data Bit 0
8	GPIO5	Pixel Data Bit 1
9	GPIO18	Pixel Data Bit 2
10	GPIO19	Pixel Data Bit 3
11	GPIO36	Pixel Data Bit 4
11	GPIO39	Pixel Data Bit 5
11	GPIO34	Pixel Data Bit 6
11	GPIO35	Pixel Data Bit 7
11	GPIO2	Camera Reset

## RGB LED

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

## MicroSD Card / J4

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

## LCD / U5

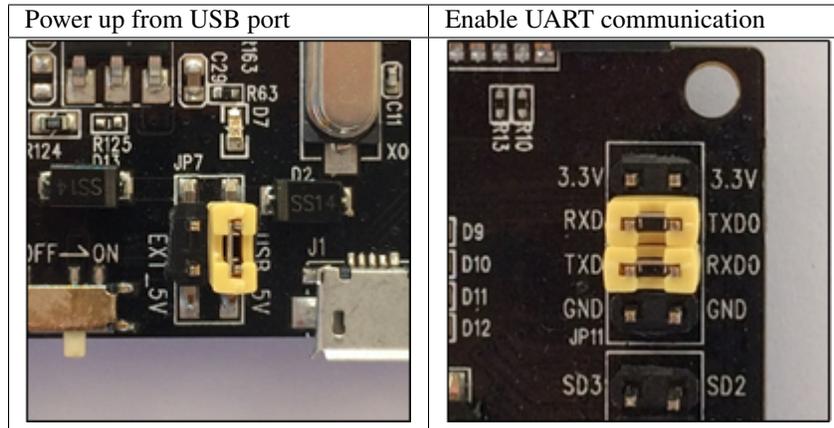
	ESP32 Pin	LCD Signal
1	GPIO18	RESET
2	GPIO19	SCL
3	GPIO21	D/C
4	GPIO22	CS
5	GPIO23	SDA
6	GPIO25	SDO
7	GPIO5	Backlight

## Start Application Development

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

## Initial Setup

Select the source of power supply for the board by setting jumper JP7. The options are either USB port or an external power supply. For this application selection of USB port is sufficient. Enable UART communication by installing jumpers on JP11. Both selections are shown in table below.



Do not install any other jumpers.

## Now to Development

To start development of applications for ESP-WROVER-KIT, proceed to section [Get Started](#), that will walk you through the following steps:

- [Setup Toolchain](#) in your PC to develop applications for ESP32 in C language
- [Connect](#) the module to the PC and verify if it is accessible
- [Build and Flash](#) an example application to the ESP32
- [Monitor](#) instantly what the application is doing

## Related Documents

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

## ESP-WROVER-KIT V2 Getting Started Guide

This user guide shows how to get started with ESP-WROVER-KIT V2 development board including description of its functionality and configuration options. For description of other versions of the ESP-WROVER-KIT check [ESP32 Hardware Reference](#).

If you like to start using this board right now, go directly to section [Start Application Development](#).

## What You Need

- 1 × ESP-WROVER-KIT V2 board
- 1 x Micro USB 2.0 Cable, Type A to Micro B
- 1 × PC loaded with Windows, Linux or Mac OS

## Overview

The ESP-WROVER-KIT is a development board produced by [Espressif](#) built around ESP32. This board is compatible with ESP32 modules, including the ESP-WROOM-32 and ESP32-WROVER. The ESP-WROVER-KIT features support for an LCD and MicroSD card. The I/O pins have been broken out from the ESP32 module for easy extension. The board carries an advanced multi-protocol USB bridge (the FTDI FT2232HL), enabling developers to use JTAG directly to debug the ESP32 through the USB interface. The development board makes secondary development easy and cost-effective.

---

**Note:** ESP-WROVER-KIT V2 integrates the ESP-WROOM-32 module by default.

---

## Functionality Overview

Block diagram below presents main components of ESP-WROVER-KIT and interconnections between components.

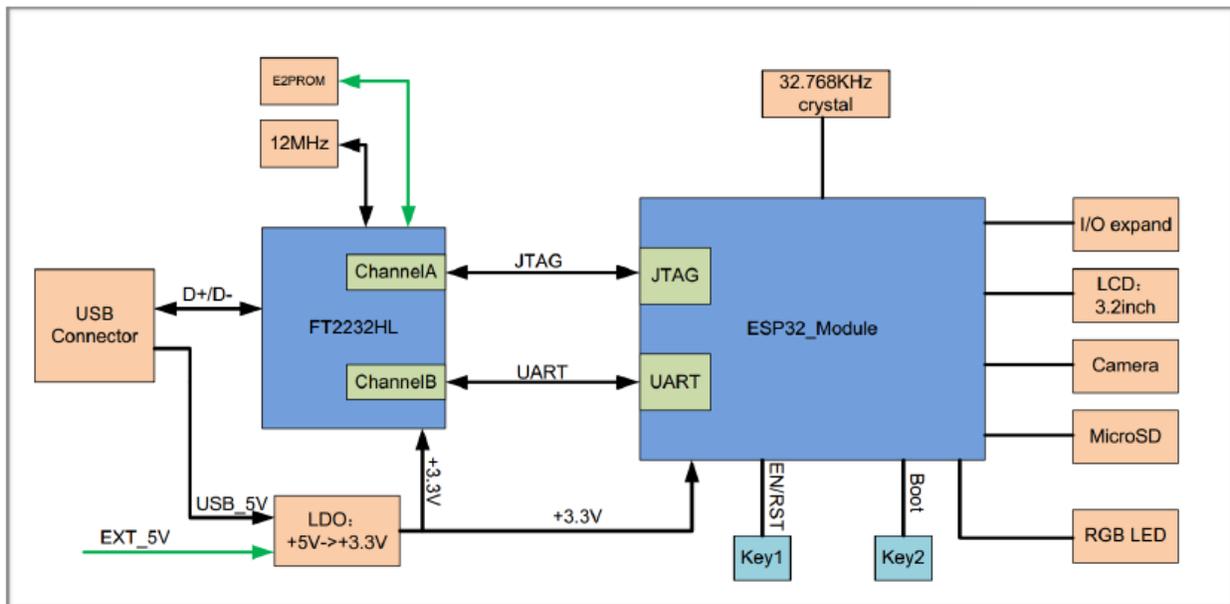


Fig. 6: ESP-WROVER-KIT block diagram

## Functional Description

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

**32.768 kHz** An external precision 32.768 kHz crystal oscillator provides the chip with a clock of low-power consumption during the Deep-sleep mode.

**ESP32 Module** ESP-WROVER-KIT is compatible with both ESP-WROOM-32 and ESP32-WROVER. The ESP32-WROVER module features all the functions of ESP-WROOM-32 and integrates an external 32-MBit PSRAM for flexible extended storage and data processing capabilities.

---

**Note:** GPIO16 and GPIO17 are used as the CS and clock signal for PSRAM. To ensure reliable performance, the two GPIOs are not broken out.

---

**CTS/RTS** Serial port flow control signals: the pins are not connected to the circuitry by default. To enable them, respective pins of JP14 must be shorted with jumpers.

**UART** Serial port: the serial TX/RX signals on FT2232HL and ESP32 are broken out to the two sides of JP11. By default, the two signals are connected with jumpers. To use the ESP32 module serial interface only, the jumpers may be removed and the module can be connected to another external serial device.

**SPI** SPI interface: the SPI interface connects to an external flash (PSRAM). To interface another SPI device, an extra CS signal is needed. If an ESP32-WROVER is being used, please note that the electrical level on the flash and SRAM is 1.8V.

**JTAG** JTAG interface: the JTAG signals on FT2232HL and ESP32 are broken out to the two sides of JP8. By default, the two signals are disconnected. To enable JTAG, shorting jumpers are required on the signals.

**FT2232** FT2232 chip is a multi-protocol USB-to-serial bridge. The FT2232 chip features USB-to-UART and USB-to-JTAG functionalities. Users can control and program the FT2232 chip through the USB interface to establish communication with ESP32.

The embedded FT2232 chip is one of the distinguishing features of the ESP-WROVER-KIT. It enhances users' convenience in terms of application development and debugging. In addition, users do not need to buy a JTAG debugger separately, which reduces the development cost, see [ESP-WROVER-KIT V2 schematic](#).

**EN** Reset button: pressing this button resets the system.

**Boot** Download button: holding down the **Boot** button and pressing the **EN** button initiates the firmware download mode. Then user can download firmware through the serial port.

**USB** USB interface. It functions as the power supply for the board and the communication interface between PC and ESP32 module.

**Power Select** Power supply selection interface: the ESP-WROVER-KIT can be powered through the USB interface or the 5V Input interface. The user can select the power supply with a jumper. More details can be found in section [Setup Options](#), jumper header JP7.

**Power Key** Power on/off button: toggling to the right powers the board on; toggling to the left powers the board off.

**5V Input** The 5V power supply interface is used as a backup power supply in case of full-load operation.

**LDO** NCP1117(1A). 5V-to-3.3V LDO. (There is an alternative pin-compatible LDO — LM317DCY, with an output current of up to 1.5A). NCP1117 can provide a maximum current of 1A. The LDO solutions are available with both fixed output voltage and variable output voltage. For details please refer to [ESP-WROVER-KIT V2 schematic](#).

**Camera** Camera interface: a standard OV7670 camera module is supported.

**RGB** Red, green and blue (RGB) light emitting diodes (LEDs), which may be controlled by pulse width modulation (PWM).

**I/O** All the pins on the ESP32 module are led out to the pin headers on the ESPWROVER-KIT. Users can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

**Micro SD Card** Micro SD card slot for data storage: when ESP32 enters the download mode, GPIO2 cannot be held high. However, a pull-up resistor is required on GPIO2 to enable the Micro SD Card. By default, GPIO2 and the pull-up resistor R153 are disconnected. To enable the SD Card, use jumpers on JP1 as shown in section *Setup Options*.

**LCD** ESP-WROVER-KIT supports mounting and interfacing a 3.2" SPI (standard 4-wire Serial Peripheral Interface) LCD, as shown on figure *ESP-WROVER-KIT board layout - back*.

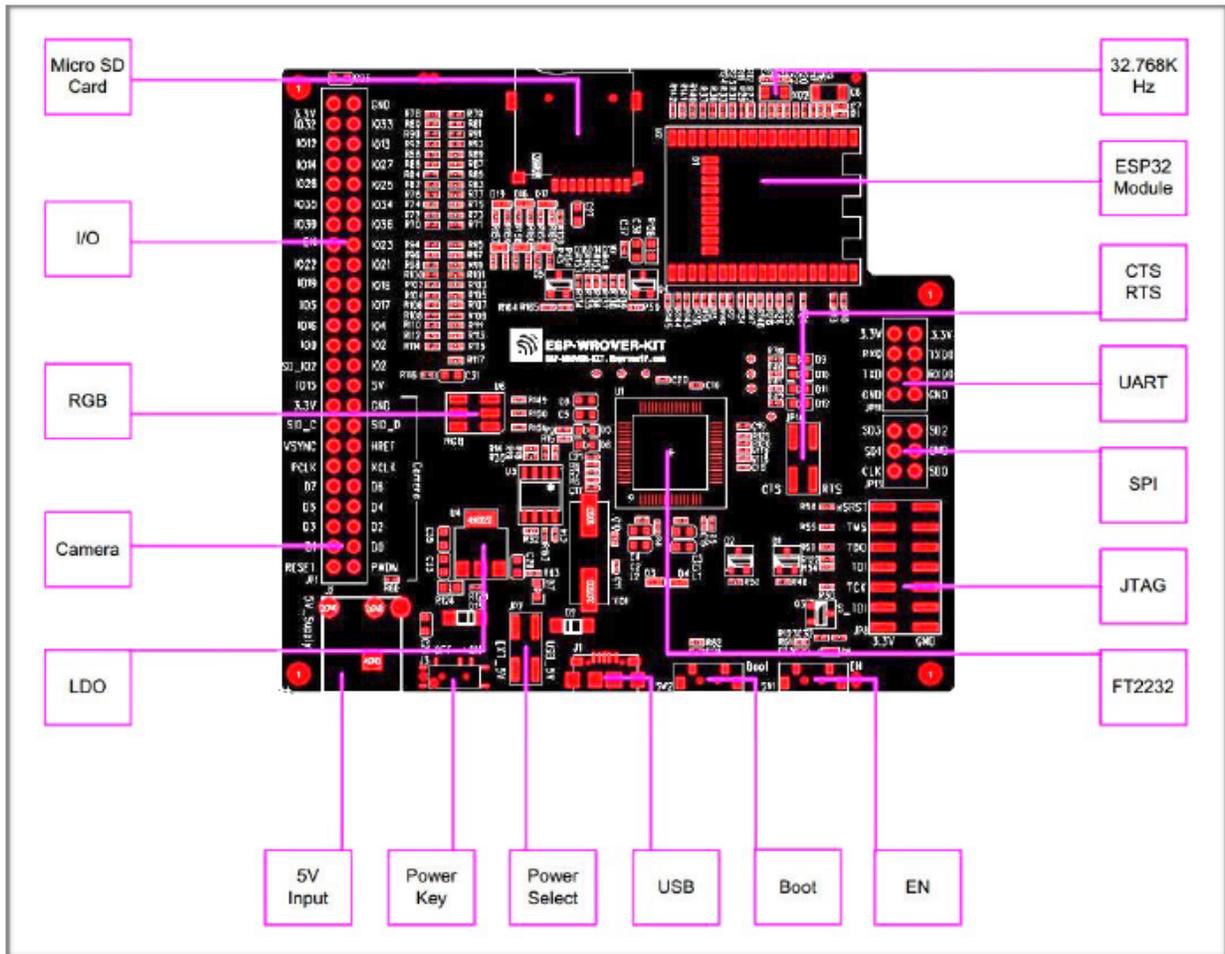


Fig. 7: ESP-WROVER-KIT board layout - front

## Setup Options

There are five jumper headers available to set up the board functionality. Typical options to select from are listed in table below.

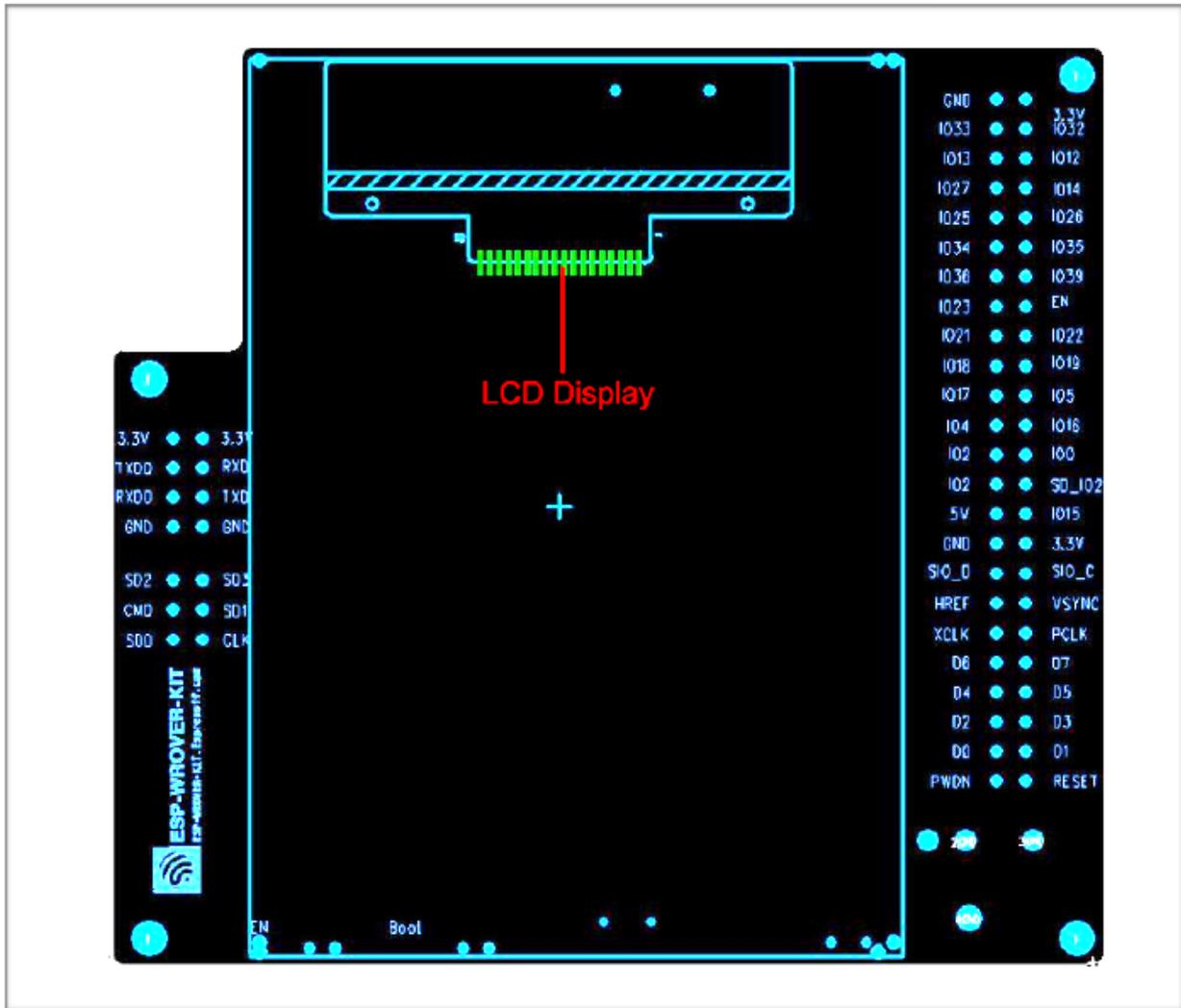
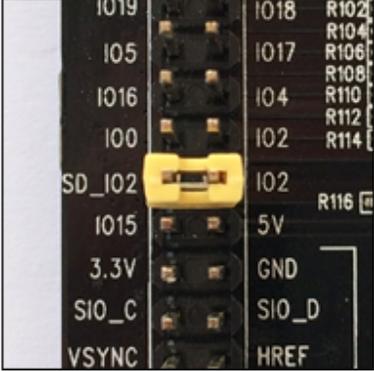
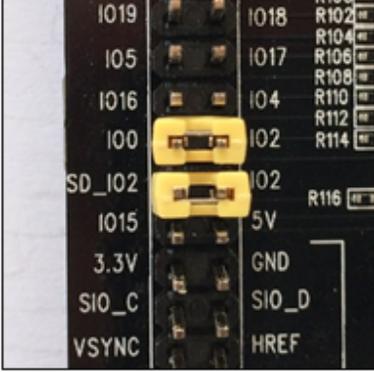
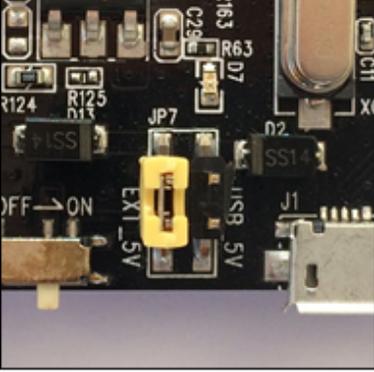
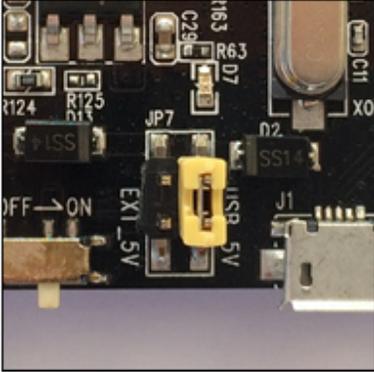
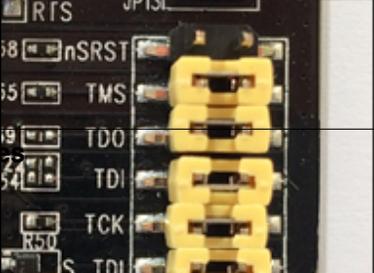


Fig. 8: ESP-WROVER-KIT board layout - back

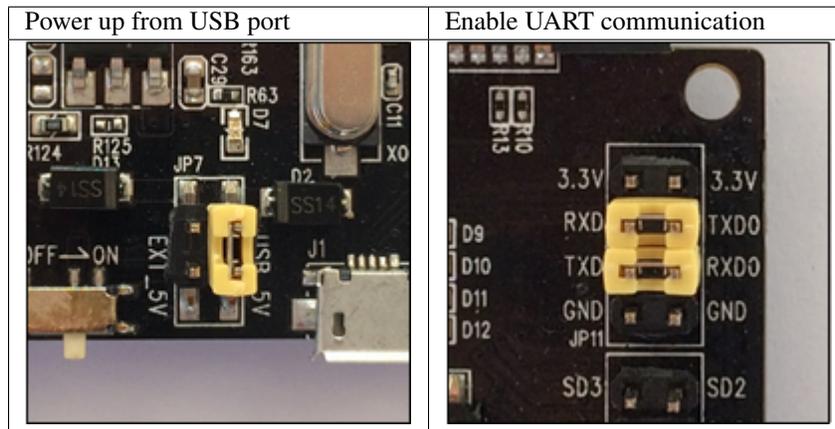
Header	Jumper Setting	Description of Functionality
JP1		Enable pull up for the Micro SD Card
JP1		Assert GPIO2 low during each download (by jumping it to GPIO0)
JP7		Power ESP-WROVER-KIT board from an external power supply
JP7		Power ESP-WROVER-KIT board from an USB port
1.3. Guide		

## Start Application Development

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

### Initial Setup

Select the source of power supply for the board by setting jumper JP7. The options are either USB port or an external power supply. For this application selection of USB port is sufficient. Enable UART communication by installing jumpers on JP11. Both selections are shown in table below.



Do not install any other jumpers.

### Now to Development

To start development of applications for ESP32-DevKitC, proceed to section *Get Started*, that will walk you through the following steps:

- *Setup Toolchain* in your PC to develop applications for ESP32 in C language
- *Connect* the module to the PC and verify if it is accessible
- *Build and Flash* an example application to the ESP32
- *Monitor* instantly what the application is doing

### Related Documents

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

### 1.3.3 ESP32-PICO-KIT V4 Getting Started Guide

This user guide shows how to get started with the ESP32-PICO-KIT V4 mini development board. For description of other versions of the ESP32-PICO-KIT check [ESP32 Hardware Reference](#).

#### What You Need

- 1 × *ESP32-PICO-KIT V4 mini development board*
- 1 × USB A / Micro USB B cable
- 1 × PC loaded with Windows, Linux or Mac OS

If you like to start using this board right now, go directly to section [Start Application Development](#).

#### Overview

ESP32-PICO-KIT V4 is a mini development board produced by [Espressif](#). At the core of this board is the ESP32-PICO-D4, a System-in-Package (SIP) module with complete Wi-Fi and Bluetooth functionalities. Comparing to other ESP32 chips, the ESP32-PICO-D4 integrates several peripheral components in one single package, that otherwise would need to be installed separately. This includes a 40 MHz crystal oscillator, 4 MB flash, filter capacitors and RF matching links in. This greatly reduces quantity and costs of additional components, subsequent assembly and testing cost, as well as overall product complexity.

The development board integrates a USB-UART Bridge circuit, allowing the developers to connect the board to a PC's USB port for downloads and debugging.

For easy interfacing, all the IO signals and system power on ESP32-PICO-D4 are led out through two rows of 20 x 0.1" pitch header pads on both sides of the development board. To make the ESP32-PICO-KIT V4 fit into mini breadboards, the header pads are populated with two rows of 17 pin headers. Remaining 2 x 3 pads grouped on each side of the board besides the antenna are not populated. The remaining 2 x 3 pin headers may be soldered later by the user.

---

**Note:** The 2 x 3 pads not populated with pin headers are internally connected to the flash memory embedded in the ESP32-PICO-D4 SIP module. For more details see module's datasheet in [Related Documents](#).

---

The board dimensions are 52 x 20.3 x 10 mm (2.1" x 0.8" x 0.4"), see section [Board Dimensions](#). An overview functional block diagram is shown below.

#### Functional Description

The following list and figure below describe key components, interfaces and controls of ESP32-PICO-KIT V4 board.

**ESP32-PICO-D4** Standard ESP32-PICO-D4 module soldered to the ESP32-PICO-KIT V4 board. The complete system of the ESP32 chip has been integrated into the SIP module, requiring only external antenna with LC matching network, decoupling capacitors and pull-up resistors for EN signals to function properly.

**LDO** 5V-to-3.3V Low dropout voltage regulator (LDO).

**USB-UART Bridge** A single chip USB-UART bridge provides up to 1 Mbps transfers rates.

**Micro USB Port** USB interface. It functions as the power supply for the board and the communication interface between PC and ESP32-PICO-KIT V4.

**5V Power On LED** This light emitting diode lits when the USB or an external 5V power supply is applied to the board. For details see schematic in [Related Documents](#).

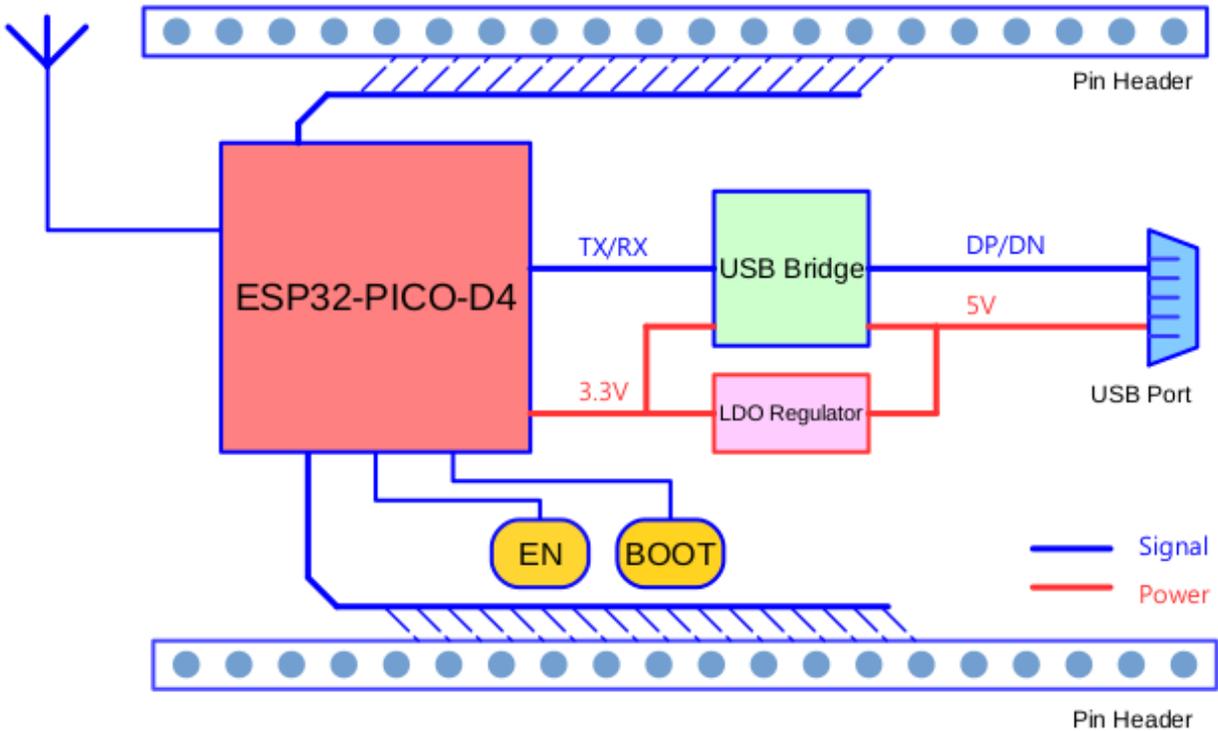


Fig. 9: ESP32-PICO-KIT V4 functional block diagram

**I/O** All the pins on ESP32-PICO-D4 are broken out to the pin headers on the board. Users can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc. For details please see section *Pin Descriptions*.

**BOOT Button** Holding down the Boot button and pressing the EN button initiates the firmware download mode. Then user can download firmware through the serial port.

**EN Button** Reset button; pressing this button resets the system.

### Power Supply Options

There following options are available to provide power supply to the ESP32-PICO-KIT V4:

1. Micro USB port, this is default power supply connection
2. 5V / GND header pins
3. 3V3 / GND header pins

**Warning:** Above options are mutually exclusive, i.e. the power supply may be provided using only one of the above options. Attempt to power the board using more than one connection at a time may damage the board and/or the power supply source.

### Start Application Development

Before powering up the ESP32-PICO-KIT V4, please make sure that the board has been received in good condition with no obvious signs of damage.

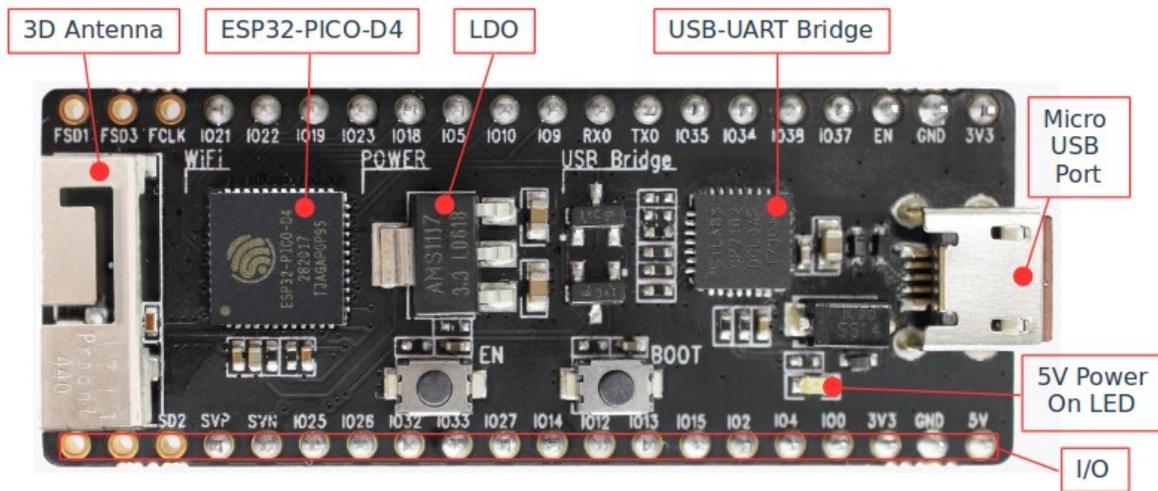


Fig. 10: ESP32-PICO-KIT V4 board layout

To start development of applications, proceed to section *Get Started*, that will walk you through the following steps:

- *Setup Toolchain* in your PC to develop applications for ESP32 in C language
- *Connect* the module to the PC and verify if it is accessible
- *Build and Flash* an example application to the ESP32
- *Monitor* instantly what the application is doing

### Pin Descriptions

The two tables below provide the **Name** and **Function** of I/O headers on both sides of the board, see *ESP32-PICO-KIT V4 board layout*. The pin numbering and header names are the same as on a schematic in *Related Documents*.

Header J2

No.	Name	Type	Function
1	FLASH_SD1 (FSD1)	I/O	GPIO8, SD_DATA1, SPIID, HS1_DATA1 (1), U2CTS
2	FLASH_SD3 (FSD3)	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0 (1), U2RTS
3	FLASH_CLK (FCLK)	I/O	GPIO11, SD_CMD, SPICS0, HS1_CMD (1), U1RTS
4	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
5	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
6	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
7	IO23	I/O	GPIO23, VSPID, HS1_STROBE
8	IO18	I/O	GPIO18, VSPICLK, HS1_DATA7
9	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
10	IO10	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
11	IO9	I/O	<b>Chapter 1. Get Started</b> GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD



Header J3

No.	Name	Type	Function
1	FLASH_CS (FCS)	I/O	GPIO16, HS1_DATA4 (1), U2RXD, EMAC_CLK_OUT
2	FLASH_SD0 (FSD0)	I/O	GPIO17, HS1_DATA5 (1), U2TXD, EMAC_CLK_OUT_180
3	FLASH_SD2 (FSD2)	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK (1), U1CTS
4	SENSOR_VP (FSVP)	I	GPIO36, ADC1_CH0, ADC_PRE_AMP (2a), RTC_GPIO0
5	SENSOR_VN (FSVN)	I	GPIO39, ADC1_CH3, ADC_PRE_AMP (2b), RTC_GPIO3
6	IO25	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
7	IO26	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
8	IO32	I/O	32K_XP (3a), ADC1_CH4, TOUCH9, RTC_GPIO9
9	IO33	I/O	32K_XN (3b), ADC1_CH5, TOUCH8, RTC_GPIO8
28			<b>Chapter 1. Get Started</b>
10	IO27	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17

### Notes to *Pin Descriptions*

1. This pin is connected to the flash pin of ESP32-PICO-D4.
2. When used as ADC\_PRE\_AMP, connect 270 pF capacitors between: (a) SENSOR\_VP and IO37, (b) SENSOR\_VN and IO38.
3. 32.768 kHz crystal oscillator: (a) input, (b) output.
4. This pin is connected to the pin of the USB bridge chip on the board.
5. The operating voltage of ESP32-PICO-KIT's embedded SPI flash is 3.3V. Therefore, the strapping pin MTDI should hold bit "0" during the module power-on reset.

### Board Dimensions

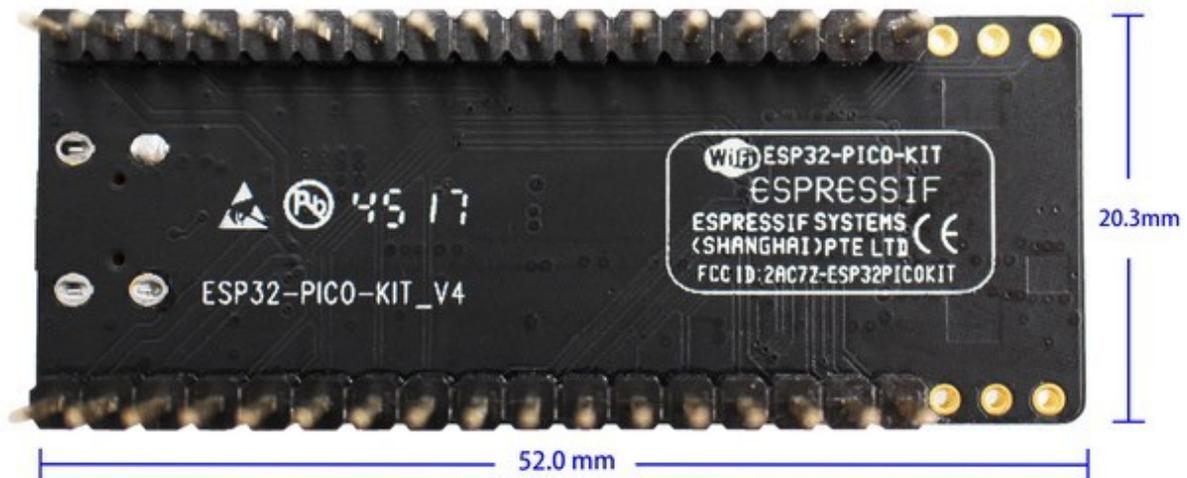


Fig. 11: ESP32-PICO-KIT V4 dimensions - back

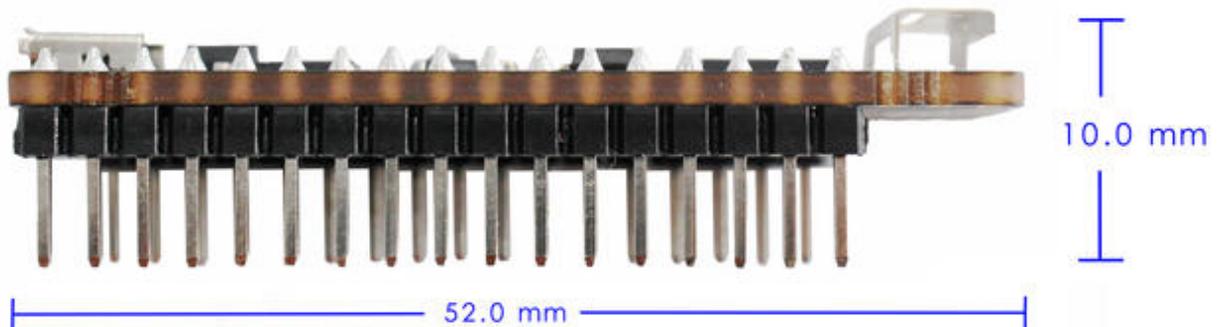


Fig. 12: ESP32-PICO-KIT V4 dimensions - side

## Related Documents

- [ESP32-PICO-KIT V4 schematic \(PDF\)](#)
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)
- [ESP32 Hardware Reference](#)

## ESP32-PICO-KIT V3 Getting Started Guide

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

## What You Need

- 1 × ESP32-PICO-KIT V3 mini development board
- 1 × USB A / Micro USB B cable
- 1 × PC loaded with Windows, Linux or Mac OS

## Overview

ESP32-PICO-KIT V3 is a mini development board based on the ESP32-PICO-D4 SIP module produced by [Espressif](#). All the IO signals and system power on ESP32-PICO-D4 are led out through two standard 20 pin x 0.1” pitch headers on both sides for easy interfacing. The development board integrates a USB-UART Bridge circuit, allowing the developers to connect the development board to a PC’s USB port for downloads and debugging.

## Functional Description

The following list and figure below describe key components, interfaces and controls of ESP32-PICO-KIT V3 board.

**ESP32-PICO-D4** Standard ESP32-PICO-D4 module soldered to the ESP32-PICO-KIT V3 board. The complete system of the ESP32 chip has been integrated into the SIP module, requiring only external antenna with LC matching network, decoupling capacitors and pull-up resistors for EN signals to function properly.

**USB-UART Bridge** A single chip USB-UART bridge provides up to 1 Mbps transfers rates.

**I/O** All the pins on ESP32-PICO-D4 are broken out to the pin headers on the board. Users can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

**Micro USB Port** USB interface. It functions as the power supply for the board and the communication interface between PC and ESP32-PICO-KIT V3.

**EN Button** Reset button; pressing this button resets the system.

**BOOT Button** Holding down the Boot button and pressing the EN button initiates the firmware download mode. Then user can download firmware through the serial port.

## Start Application Development

Before powering up the ESP32-PICO-KIT V3, please make sure that the board has been received in good condition with no obvious signs of damage.

To start development of applications, proceed to section [Get Started](#), that will walk you through the following steps:

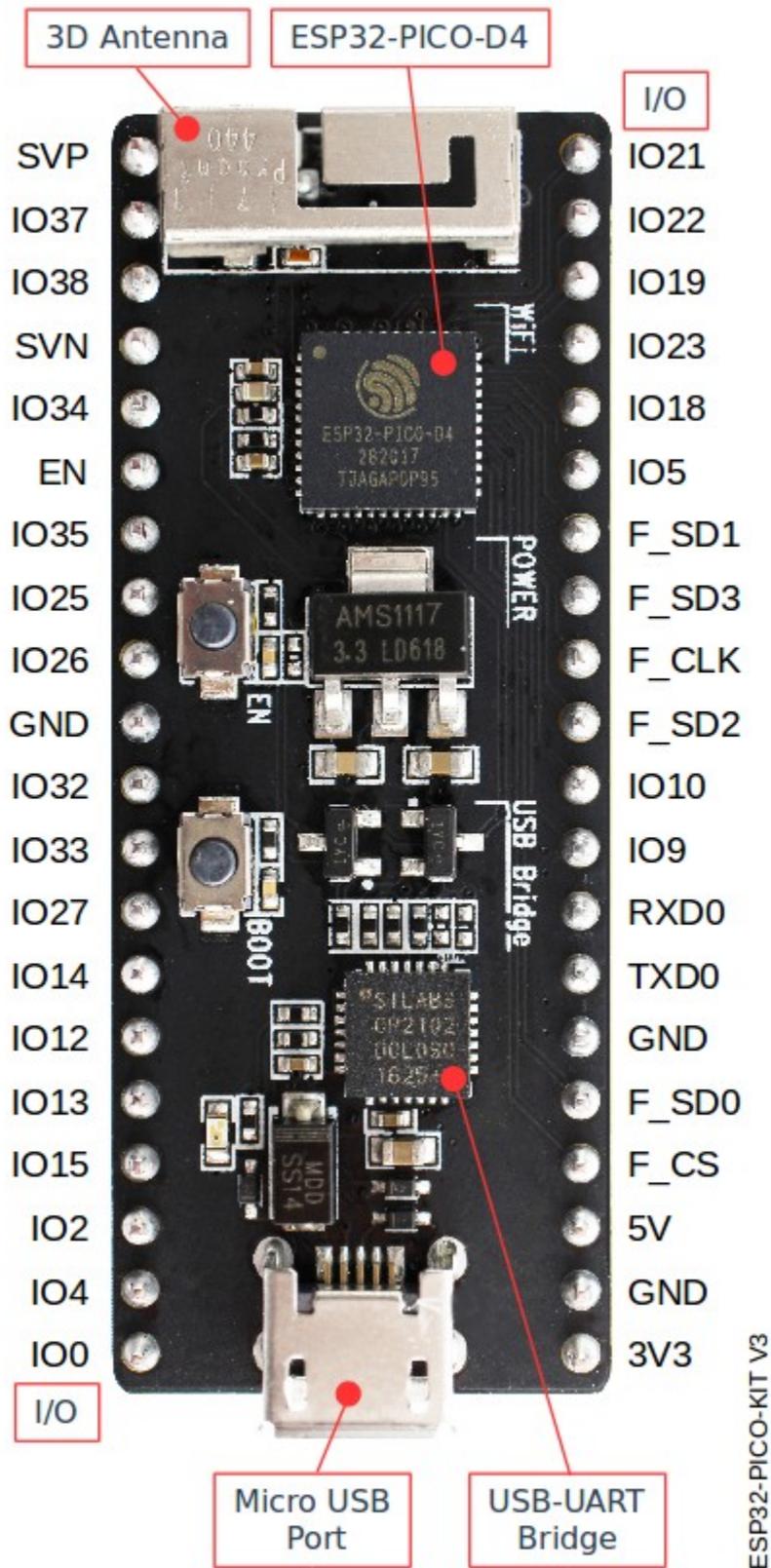


Fig. 13: ESP32-PICO-KIT V3 board layout

- *Setup Toolchain* in your PC to develop applications for ESP32 in C language
- *Connect* the module to the PC and verify if it is accessible
- *Build and Flash* an example application to the ESP32
- *Monitor* instantly what the application is doing

## Related Documents

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

If you have different board, move to sections below.

## 1.4 Setup Toolchain

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

### 1.4.1 Standard Setup of Toolchain for Windows

#### Introduction

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

#### Toolchain Setup

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

[https://dl.espressif.com/dl/esp32\\_win32\\_msys2\\_environment\\_and\\_toolchain-20180110.zip](https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_toolchain-20180110.zip)

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

#### Check it Out

Open a MSYS2 MINGW32 terminal window by running `C:\msys32\mingw32.exe`. The environment in this window is a bash shell.

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

#### Next Steps

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

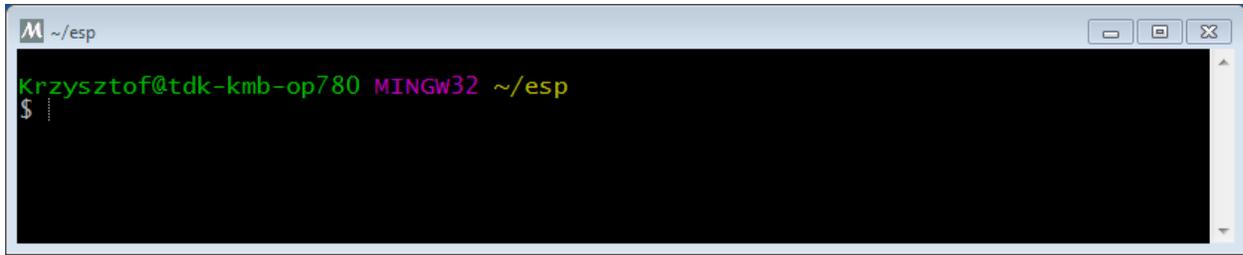


Fig. 14: MSYS2 MINGW32 shell window

## Updating The Environment

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

- Take the old MSYS2 environment (ie `C:\msys32`) and move/rename it to a different directory (ie `C:\msys32_old`).
- Download the new precompiled environment using the steps above.
- Unzip the new MSYS2 environment to `C:\msys32` (or another location).
- Find the old `C:\msys32_old\home` directory and move this into `C:\msys32`.
- You can now delete the `C:\msys32_old` directory if you no longer need it.

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

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

## Related Documents

### Setup Windows Toolchain from Scratch

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

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

### Configure Toolchain & Environment from Scratch

This process involves installing `MSYS2`, then installing the `MSYS2` and Python packages which ESP-IDF uses, and finally downloading and installing the Xtensa toolchain.

- Navigate to the `MSYS2` installer page and download the `msys2-i686-xxxxxxx.exe` installer executable (we only support a 32-bit MSYS environment, it works on both 32-bit and 64-bit Windows.) At time of writing, the latest installer is `msys2-i686-20161025.exe`.
- Run through the installer steps. **Uncheck the “Run MSYS2 32-bit now” checkbox at the end.**
- Once the installer exits, open Start Menu and find “MSYS2 MinGW 32-bit” to run the terminal.

*(Why launch this different terminal? MSYS2 has the concept of different kinds of environments. The default “MSYS” environment is Cygwin-like and uses a translation layer for all Windows API calls. We need the “MinGW” environment in order to have a native Python which supports COM ports.)*

- The ESP-IDF repository on github contains a script in the tools directory titled `windows_install_prerequisites.sh`. If you haven’t got a local copy of the ESP-IDF yet, that’s OK - you can just download that one file in Raw format from here: [tools/windows/windows\\_install\\_prerequisites.sh](#). Save it somewhere on your computer.
- Type the path to the shell script into the MSYS2 terminal window. You can type it as a normal Windows path, but use forward-slashes instead of back-slashes. ie: `C:/Users/myuser/Downloads/windows_install_prerequisites.sh`. You can read the script beforehand to check what it does.
- The `windows_install_prerequisites.sh` script will download and install packages for ESP-IDF support, and the ESP32 toolchain.

## Troubleshooting

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

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

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

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

## MSYS2 Mirrors in China

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

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

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

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

Add these lines at the top of `mirrorlist.msys2`:

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

## HTTP Proxy

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

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

Or with credentials:

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

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

### Alternative Setup: Just download a toolchain

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

<https://dl.espressif.com/dl/xtensa-esp32-elf-win32-1.22.0-80-g6c4433a-5.2.0.zip>

---

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

---

---

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

---

### Next Steps

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

### Updating The Environment

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

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

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

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

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

## 1.4.2 Standard Setup of Toolchain for Linux

### Install Prerequisites

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

- CentOS 7:

```
sudo yum install git wget make ncurses-devel flex bison gperf python pyserial
```

- Ubuntu and Debian:

```
sudo apt-get install git wget make libncurses-dev flex bison gperf python python-serial
```

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python2-pyserial
```

## Toolchain Setup

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

- for 64-bit Linux:

<https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz>

- for 32-bit Linux:

<https://dl.espressif.com/dl/xtensa-esp32-elf-linux32-1.22.0-80-g6c4433a-5.2.0.tar.gz>

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

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz
```

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

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

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

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

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

Then when you need the toolchain you can type get\_esp32 on the command line and the toolchain will be added to your PATH.

---

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

---

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

```
printenv PATH
```

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

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

(continues on next page)

(continued from previous page)

---

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

### Permission issues `/dev/ttyUSB0`

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

### Arch Linux Users

To run the precompiled gdb (xtensa-esp32-elf-gdb) in Arch Linux requires ncurses 5, but Arch uses ncurses 6.

Backwards compatibility libraries are available in AUR for native and lib32 configurations:

- <https://aur.archlinux.org/packages/ncurses5-compat-libs/>
- <https://aur.archlinux.org/packages/lib32-ncurses5-compat-libs/>

Before installing these packages you might need to add the author's public key to your keyring as described in the "Comments" section at the links above.

Alternatively, use crosstool-NG to compile a gdb that links against ncurses 6.

### Next Steps

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

### Related Documents

#### Setup Linux Toolchain from Scratch

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

#### Install Prerequisites

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

- Ubuntu and Debian:

```
sudo apt-get install git wget make libncurses-dev flex bison gperf python python-  
↪serial
```

- Arch:

```
sudo pacman -S --needed gcc git make ncurses flex bison gperf python2-pyserial
```

## Compile the Toolchain from Source

- Install dependencies:

- CentOS 7:

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

- Ubuntu pre-16.04:

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

- Ubuntu 16.04:

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

- Debian:

```
TODO
```

- Arch:

```
TODO
```

Download `crosstool-NG` and build it:

```
cd ~/esp  
git clone -b xtensa-1.22.x https://github.com/espressif/crosstool-NG.git  
cd crosstool-NG  
./bootstrap && ./configure --enable-local && make install
```

Build the toolchain:

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

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

## Next Steps

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

### 1.4.3 Standard Setup of Toolchain for Mac OS

#### Install Prerequisites

- install pip:

```
sudo easy_install pip
```

- install pyserial:

```
sudo pip install pyserial
```

## Toolchain Setup

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

<https://dl.espressif.com/dl/xtensa-esp32-elf-osx-1.22.0-80-g6c4433a-5.2.0.tar.gz>

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

```
mkdir -p ~/esp
cd ~/esp
tar -xzf ~/Downloads/xtensa-esp32-elf-osx-1.22.0-80-g6c4433a-5.2.0.tar.gz
```

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

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

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

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

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

Then when you need the toolchain you can type get\_esp32 on the command line and the toolchain will be added to your PATH.

## Next Steps

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

## Related Documents

### Setup Toolchain for Mac OS from Scratch

#### Install Prerequisites

- install pip:

```
sudo easy_install pip
```

- install pyserial:

```
sudo pip install pyserial
```

#### Compile the Toolchain from Source

- Install dependencies:

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

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

- with homebrew:

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

Create a case-sensitive filesystem image:

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

Mount it:

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

Create a symlink to your work directory:

```
cd ~/esp  
ln -s /Volumes/ctng crosstool-NG ↵
```

Download crosstool-NG and build it:

```
cd ~/esp  
git clone -b xtensa-1.22.x https://github.com/espressif/crosstool-NG.git  
cd crosstool-NG  
./bootstrap && ./configure --enable-local && make install ↵
```

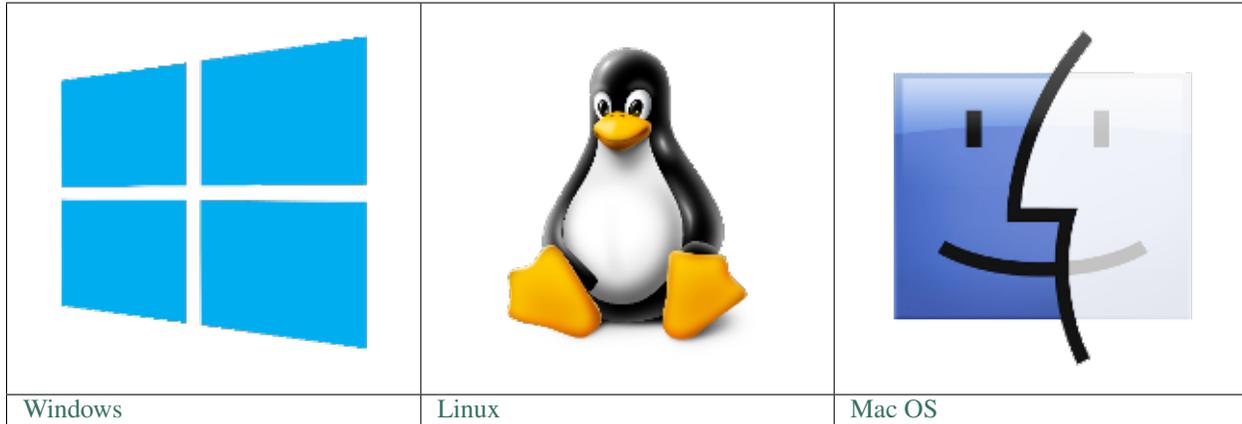
Build the toolchain:

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

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

## Next Steps

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



**Note:** We are using `~/esp` directory to install the prebuilt toolchain, ESP-IDF and sample applications. You can use different directory, but need to adjust respective commands.

Depending on your experience and preferences, instead of using a prebuilt toolchain, you may want to customize your environment. To set up the system your own way go to section [Customized Setup of Toolchain](#).

Once you are done with setting up the toolchain then go to section [Get ESP-IDF](#).

## 1.5 Get ESP-IDF

Besides the toolchain (that contains programs to compile and build the application), you also need ESP32 specific API / libraries. They are provided by Espressif in [ESP-IDF repository](#).

To obtain a local copy: open terminal, navigate to the directory you want to put ESP-IDF, and clone the repository using `git clone` command:

```
cd ~/esp
git clone -b release/v3.0 --recursive https://github.com/espressif/esp-idf.git
```

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

**Note:** The `git clone` option `-b release/v3.0` tells git to clone the branch in the ESP-IDF repository corresponding to this version of the documentation.

**Note:** GitHub's "Download zip file" feature does not work with ESP-IDF, a `git clone` is required. As a fallback, [Stable version](#) can be installed without Git.

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

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

```
cd ~/esp/esp-idf
git submodule update --init --recursive
```

## 1.6 Setup Path to ESP-IDF

The toolchain programs access ESP-IDF using `IDF_PATH` environment variable. This variable should be set up on your PC, otherwise projects will not build. Setting may be done manually, each time PC is restarted. Another option is to set up it permanently by defining `IDF_PATH` in user profile. To do so, follow instructions specific to *Windows* , *Linux and MacOS* in section *Add IDF\_PATH to User Profile*.

## 1.7 Start a Project

Now you are ready to prepare your application for ESP32. To start off quickly, we will use `get-started/hello_world` project from `examples` directory in IDF.

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

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

You can also find a range of example projects under the `examples` directory in ESP-IDF. These example project directories can be copied in the same way as presented above, to begin your own projects.

---

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

---

## 1.8 Connect

You are almost there. To be able to proceed further, connect ESP32 board to PC, check under what serial port the board is visible and verify if serial communication works. If you are not sure how to do it, check instructions in section *Establish Serial Connection with ESP32*. Note the port number, as it will be required in the next step.

## 1.9 Configure

Being in terminal window, go to directory of `hello_world` application by typing `cd ~/esp/hello_world`. Then start project configuration utility `menuconfig`:

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

If previous steps have been done correctly, the following menu will be displayed:

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

---

**Note:** On Windows, serial ports have names like COM1. On MacOS, they start with `/dev/cu..` On Linux, they start with `/dev/tty`. (See *Establish Serial Connection with ESP32* for full details.)

---

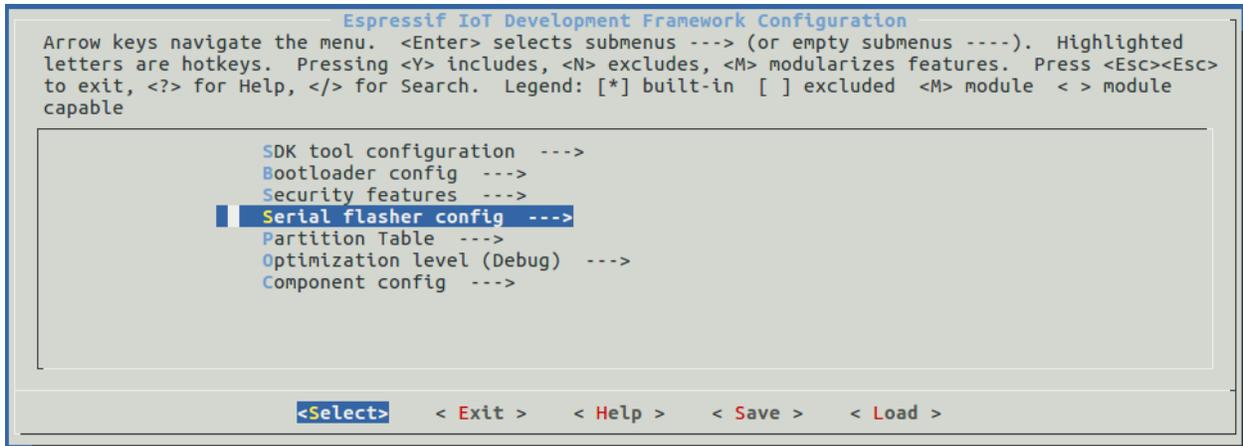


Fig. 15: Project configuration - Home window

Here are couple of tips on navigation and use of menuconfig:

- Use up & down arrow keys to navigate the menu.
- Use Enter key to go into a submenu, Escape key to go out or to exit.
- Type ? to see a help screen. Enter key exits the help screen.
- Use Space key, or Y and N keys to enable (Yes) and disable (No) configuration items with checkboxes “[\*]”
- Pressing ? while highlighting a configuration item displays help about that item.
- Type / to search the configuration items.

---

**Note:** If you are Arch Linux user, navigate to SDK tool configuration and change the name of Python 2 interpreter from python to python2.

---

## 1.10 Build and Flash

Now you can build and flash the application. Run:

```
make flash
```

This will compile the application and all the ESP-IDF components, generate bootloader, partition table, and application binaries, and flash these binaries to your ESP32 board.

```
esptool.py v2.0-beta2
Flashing binaries to serial port /dev/ttyUSB0 (app at offset 0x10000)...
esptool.py v2.0-beta2
Connecting.....____
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 921600
Changed.
Attaching SPI flash...
Configuring flash size...
```

(continues on next page)

(continued from previous page)

```

Auto-detected Flash size: 4MB
Flash params set to 0x0220
Compressed 11616 bytes to 6695...
Wrote 11616 bytes (6695 compressed) at 0x00001000 in 0.1 seconds (effective 920.5
↪kbit/s)...
Hash of data verified.
Compressed 408096 bytes to 171625...
Wrote 408096 bytes (171625 compressed) at 0x00010000 in 3.9 seconds (effective 847.3
↪kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 82...
Wrote 3072 bytes (82 compressed) at 0x00008000 in 0.0 seconds (effective 8297.4 kbit/
↪s)...
Hash of data verified.

Leaving...
Hard resetting...

```

If there are no issues, at the end of build process, you should see messages describing progress of loading process. Finally, the end module will be reset and “hello\_world” application will start.

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

## 1.11 Monitor

To see if “hello\_world” application is indeed running, type `make monitor`. This command is launching *IDF Monitor* application:

```

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

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

```

Several lines below, after start up and diagnostic log, you should see “Hello world!” printed out by the application.

```

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

```

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

**Note:** If instead of the messages above, you see a random garbage similar to:

```

e) (Xn@y.! (PW+) Hn9a/9!t5P~keea5jA
~zYY(1,1 e) (Xn@y.!DrzY(jpi|+z5Ymvp

```

or monitor fails shortly after upload, your board is likely using 26MHz crystal, while the ESP-IDF assumes default of 40MHz. Exit the monitor, go back to the *menuconfig*, change *ESP32\_XTAL\_FREQ\_SEL* to 26MHz, then *build and flash* the application again.

---

To execute `make flash` and `make monitor` in one go, type `make flash monitor`. Check section *IDF Monitor* for handy shortcuts and more details on using this application.

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

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

## 1.12 Updating ESP-IDF

After some time of using ESP-IDF, you may want to update it to take advantage of new features or bug fixes. The simplest way to do so is by deleting existing `esp-idf` folder and cloning it again, exactly as when doing initial installation described in sections *Get ESP-IDF*.

If downloading to a new path, remember to *Add IDF\_PATH to User Profile* so that the toolchain scripts know where to find the ESP-IDF in its release specific location.

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

## 1.13 Related Documents

### 1.13.1 Add IDF\_PATH to User Profile

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

#### Windows

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

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

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

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

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

```
printenv IDF_PATH
```

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

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

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

If you got here from section *Setup Path to ESP-IDF*, while installing s/w for ESP32 development, then go back to section *Start a Project*.

### Linux and MacOS

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

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

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

---

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

---

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

```
printenv IDF_PATH
```

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

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

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

If you got here from section *Setup Path to ESP-IDF*, while installing s/w for ESP32 development, then go back to section *Start a Project*.

## 1.13.2 Establish Serial Connection with ESP32

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

### Connect ESP32 to PC

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

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

- [ESP32-PICO-KIT and ESP32-DevKitC - CP210x USB to UART Bridge VCP Drivers](#)
- [ESP32-WROVER-KIT and ESP32 Demo Board - FTDI Virtual COM Port Drivers](#)

Above drivers are primarily for reference. They should already be bundled with the operating system and installed automatically once one of listed boards is connected to the PC.

## Check port on Windows

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

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

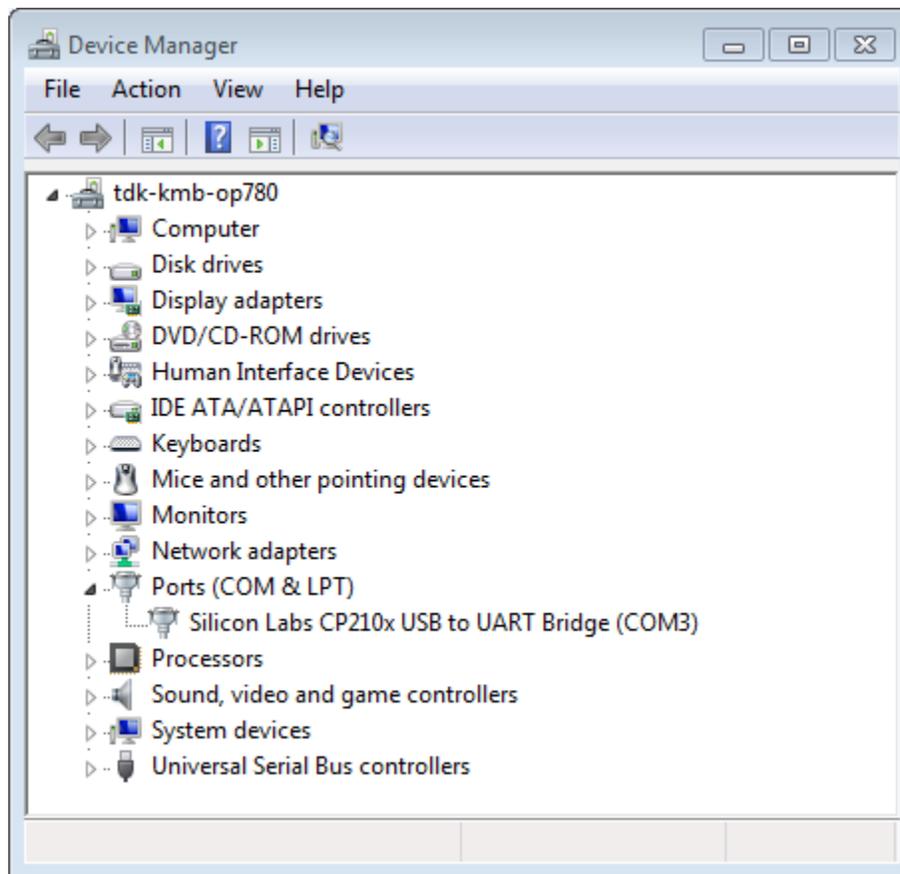


Fig. 16: USB to UART bridge of ESP32-DevKitC in Windows Device Manager

## Check port on Linux and MacOS

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

Linux

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

MacOS

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

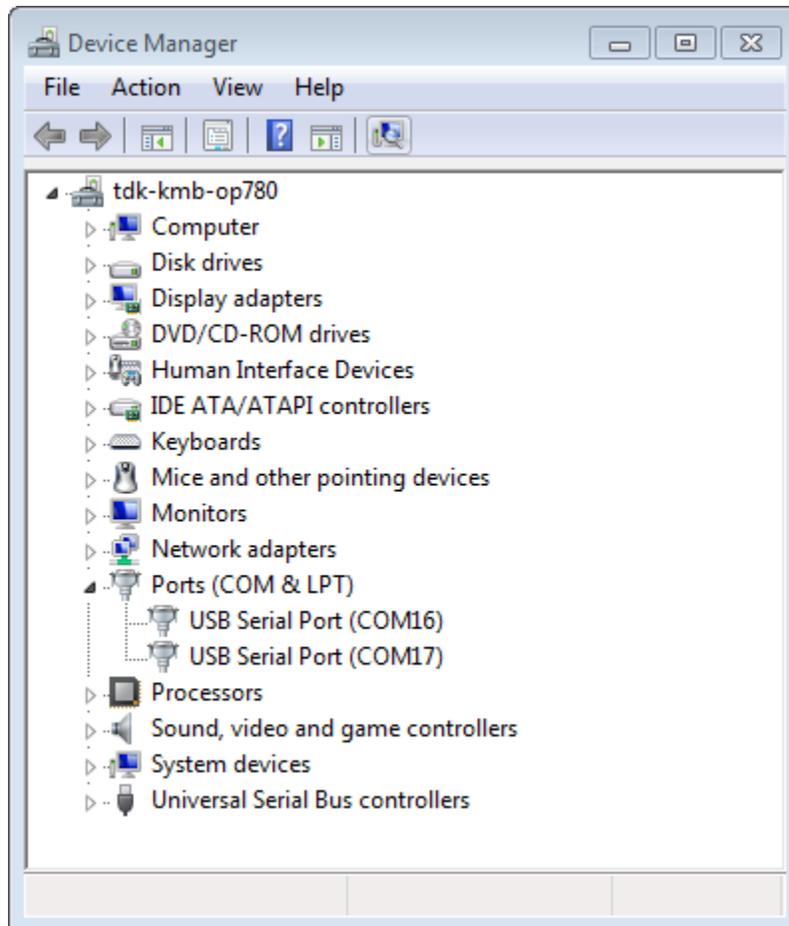


Fig. 17: Two USB Serial Ports of ESP-WROVER-KIT in Windows Device Manager

## Adding user to dialout on Linux

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

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

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

## Verify serial connection

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

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

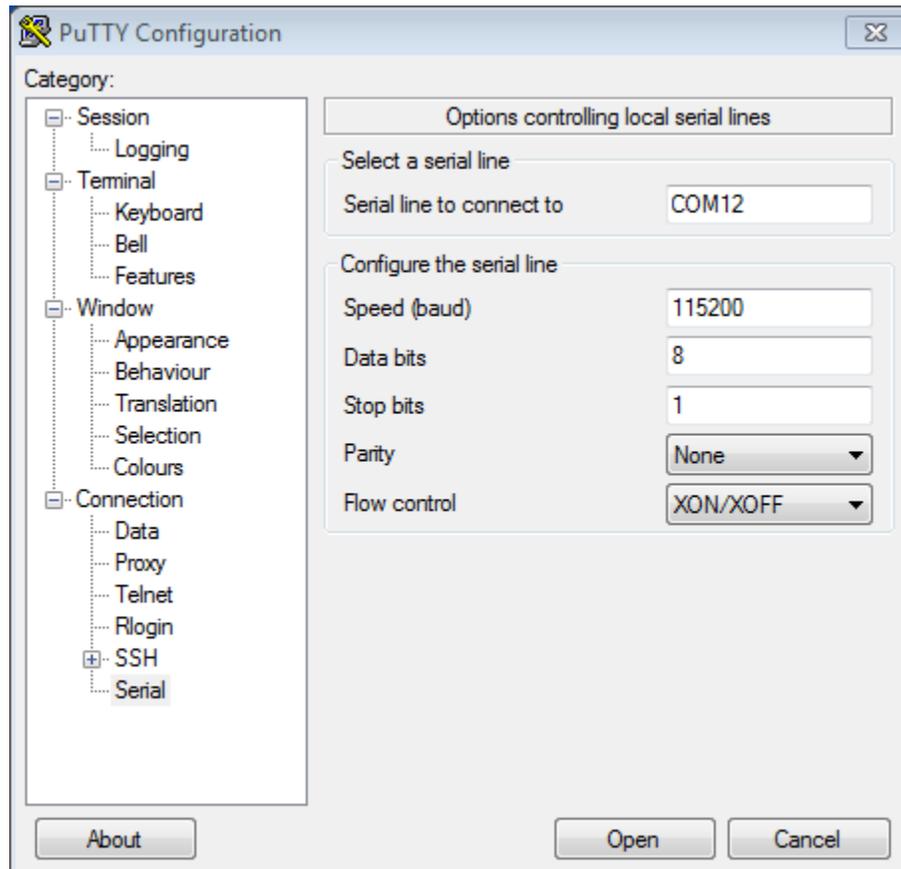


Fig. 18: Setting Serial Communication in PuTTY on Windows

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

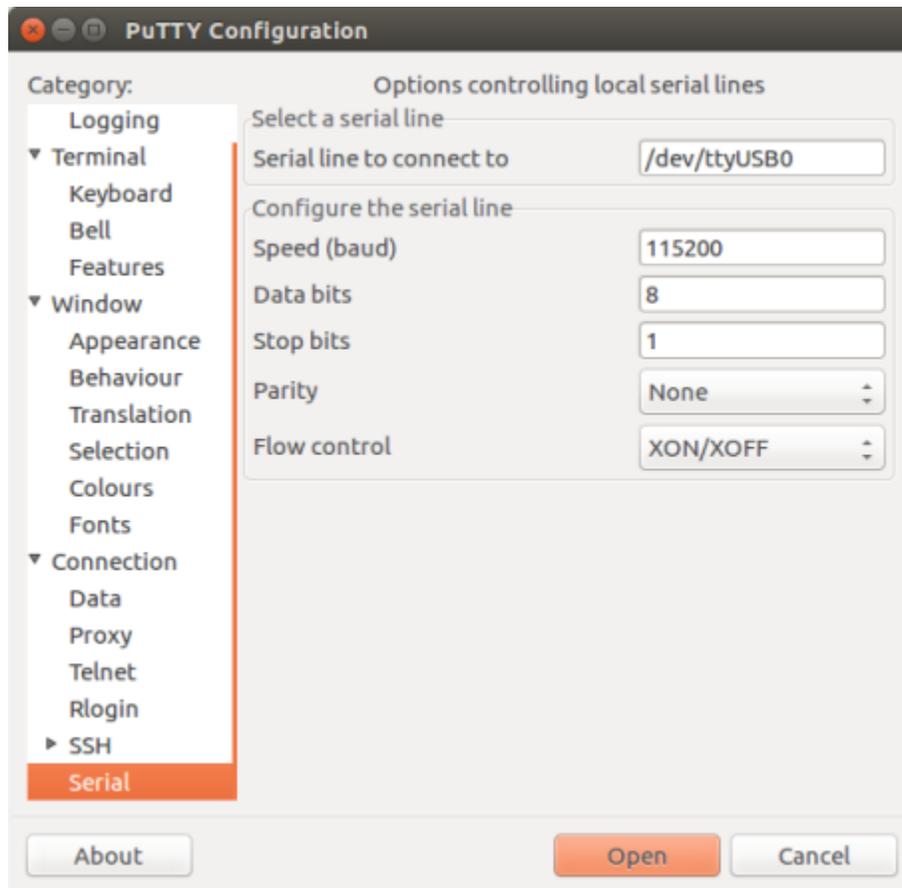


Fig. 19: Setting Serial Communication in PuTTY on Linux

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

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

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

...
```

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

---

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

---

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

---

If you got here from section [Connect](#) when installing s/w for ESP32 development, then go back to section [Configure](#).

### 1.13.3 Build and Flash with Make

#### Finding a project

As well as the `esp-idf-template` project, ESP-IDF comes with some example projects on github in the `examples` directory.

Once you've found the project you want to work with, change to its directory and you can configure and build it.

#### Configuring your project

```
make menuconfig
```

#### Compiling your project

```
make all
```

... will compile app, bootloader and generate a partition table based on the config.

## Flashing your project

When `make all` finishes, it will print a command line to use `esptool.py` to flash the chip. However you can also do this from `make` by running:

```
make flash
```

This will flash the entire project (app, bootloader and partition table) to a new chip. The settings for serial port flashing can be configured with `make menuconfig`.

You don't need to run `make all` before running `make flash`, `make flash` will automatically rebuild anything which needs it.

## Compiling & Flashing Just the App

After the initial flash, you may just want to build and flash just your app, not the bootloader and partition table:

- `make app` - build just the app.
- `make app-flash` - flash just the app.

`make app-flash` will automatically rebuild the app if it needs it.

There's no downside to reflashing the bootloader and partition table each time, if they haven't changed.

## The Partition Table

Once you've compiled your project, the "build" directory will contain a binary file with a name like "my\_app.bin". This is an ESP32 image binary that can be loaded by the bootloader.

A single ESP32's flash can contain multiple apps, as well as many kinds of data (calibration data, filesystems, parameter storage, etc). For this reason, a partition table is flashed to offset 0x8000 in the flash.

Each entry in the partition table has a name (label), type (app, data, or something else), subtype and the offset in flash where the partition is loaded.

The simplest way to use the partition table is to `make menuconfig` and choose one of the simple predefined partition tables:

- "Single factory app, no OTA"
- "Factory app, two OTA definitions"

In both cases the factory app is flashed at offset 0x10000. If you `make partition_table` then it will print a summary of the partition table.

For more details about *partition tables* and how to create custom variations, view the *documentation*.

### 1.13.4 Build and Flash with Eclipse IDE

#### Installing Eclipse IDE

The Eclipse IDE gives you a graphical integrated development environment for writing, compiling and debugging ESP-IDF projects.

- Start by installing the esp-idf for your platform (see files in this directory with steps for Windows, OS X, Linux).

- We suggest building a project from the command line first, to get a feel for how that process works. You also need to use the command line to configure your esp-idf project (via `make menuconfig`), this is not currently supported inside Eclipse.
- Download the Eclipse Installer for your platform from [eclipse.org](http://eclipse.org).
- When running the Eclipse Installer, choose “Eclipse for C/C++ Development” (in other places you’ll see this referred to as CDT.)

## Windows Users

Using ESP-IDF with Eclipse on Windows requires different configuration steps. *See the [Eclipse IDE on Windows guide](#).*

## Setting up Eclipse

Once your new Eclipse installation launches, follow these steps:

### Import New Project

- Eclipse makes use of the Makefile support in ESP-IDF. This means you need to start by creating an ESP-IDF project. You can use the `idf-template` project from github, or open one of the examples in the `esp-idf` examples subdirectory.
- Once Eclipse is running, choose File -> Import...
- In the dialog that pops up, choose “C/C++” -> “Existing Code as Makefile Project” and click Next.
- On the next page, enter “Existing Code Location” to be the directory of your IDF project. Don’t specify the path to the ESP-IDF directory itself (that comes later). The directory you specify should contain a file named “Makefile” (the project Makefile).
- On the same page, under “Toolchain for Indexer Settings” choose “Cross GCC”. Then click Finish.

## Project Properties

- The new project will appear under Project Explorer. Right-click the project and choose Properties from the context menu.
- Click on the “Environment” properties page under “C/C++ Build”. Click “Add...” and enter name `BATCH_BUILD` and value `1`.
- Click “Add...” again, and enter name `IDF_PATH`. The value should be the full path where ESP-IDF is installed.
- Edit the `PATH` environment variable. Keep the current value, and append the path to the Xtensa toolchain that will be installed as part of IDF setup (`something/xtensa-esp32-elf/bin`) if this is not already listed on the `PATH`.
- On macOS, add a `PYTHONPATH` environment variable and set it to `/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages`. This is so that the system Python, which has `pyserial` installed as part of the setup steps, overrides any built-in Eclipse Python.

Navigate to “C/C++ General” -> “Preprocessor Include Paths” property page:

- Click the “Providers” tab

- In the list of providers, click “CDT Cross GCC Built-in Compiler Settings”. Change “Command to get compiler specs” to `xtensa-esp32-elf-gcc ${FLAGS} -E -P -v -dD "${INPUTS} ”`.
- In the list of providers, click “CDT GCC Build Output Parser” and change the “Compiler command pattern” to `xtensa-esp32-elf- (gcc|g\+\+|c\+\+|cc|cpp|clang)`

Navigate to “C/C++ General” -> “Indexer” property page:

- Check “Enable project specific settings” to enable the rest of the settings on this page.
- Uncheck “Allow heuristic resolution of includes”. When this option is enabled Eclipse sometimes fails to find correct header directories.

### Building in Eclipse

Before your project is first built, Eclipse may show a lot of errors and warnings about undefined values. This is because some source files are automatically generated as part of the esp-idf build process. These errors and warnings will go away after you build the project.

- Click OK to close the Properties dialog in Eclipse.
- Outside Eclipse, open a command line prompt. Navigate to your project directory, and run `make menuconfig` to configure your project’s esp-idf settings. This step currently has to be run outside Eclipse.

*If you try to build without running a configuration step first, esp-idf will prompt for configuration on the command line - but Eclipse is not able to deal with this, so the build will hang or fail.*

- Back in Eclipse, choose Project -> Build to build your project.

**TIP:** If your project had already been built outside Eclipse, you may need to do a Project -> Clean before choosing Project -> Build. This is so Eclipse can see the compiler arguments for all source files. It uses these to determine the header include paths.

### Flash from Eclipse

You can integrate the “make flash” target into your Eclipse project to flash using `esptool.py` from the Eclipse UI:

- Right-click your project in Project Explorer (important to make sure you select the project, not a directory in the project, or Eclipse may find the wrong Makefile.)
- Select Make Targets -> Create from the context menu.
- Type “flash” as the target name. Leave the other options as their defaults.
- Now you can use Project -> Make Target -> Build (Shift+F9) to build the custom flash target, which will compile and flash the project.

Note that you will need to use “make menuconfig” to set the serial port and other config options for flashing. “make menuconfig” still requires a command line terminal (see the instructions for your platform.)

Follow the same steps to add `bootloader` and `partition_table` targets, if necessary.

### Related Documents

#### Eclipse IDE on Windows

Configuring Eclipse on Windows requires some different steps. The full configuration steps for Windows are shown below.

(For OS X and Linux instructions, see the [Eclipse IDE page](#).)

## Installing Eclipse IDE

Follow the steps under [Installing Eclipse IDE](#) for all platforms.

## Setting up Eclipse on Windows

Once your new Eclipse installation launches, follow these steps:

### Import New Project

- Eclipse makes use of the Makefile support in ESP-IDF. This means you need to start by creating an ESP-IDF project. You can use the `idf-template` project from github, or open one of the examples in the `esp-idf` examples subdirectory.
- Once Eclipse is running, choose File -> Import...
- In the dialog that pops up, choose “C/C++” -> “Existing Code as Makefile Project” and click Next.
- On the next page, enter “Existing Code Location” to be the directory of your IDF project. Don’t specify the path to the ESP-IDF directory itself (that comes later). The directory you specify should contain a file named “Makefile” (the project Makefile).
- On the same page, under “Toolchain for Indexer Settings” uncheck “Show only available toolchains that support this platform”.
- On the extended list that appears, choose “Cygwin GCC”. Then click Finish.

*Note: you may see warnings in the UI that Cygwin GCC Toolchain could not be found. This is OK, we’re going to reconfigure Eclipse to find our toolchain.*

### Project Properties

- The new project will appear under Project Explorer. Right-click the project and choose Properties from the context menu.
- Click on the “C/C++ Build” properties page (top-level):
  - Uncheck “Use default build command” and enter this for the custom build command: `python ${IDF_PATH}/tools/windows/eclipse_make.py`
- Click on the “Environment” properties page under “C/C++ Build”:
  - Click “Add...” and enter name `BATCH_BUILD` and value `1`.
  - Click “Add...” again, and enter name `IDF_PATH`. The value should be the full path where ESP-IDF is installed. The `IDF_PATH` directory should be specified using forwards slashes not backslashes, ie `C:/Users/MyUser/Development/esp-idf`.
  - Edit the `PATH` environment variable. Delete the existing value and replace it with `C:\msys32\usr\bin;C:\msys32\mingw32\bin;C:\msys32\opt\xtensa-esp32-elf\bin` (If you installed `msys32` to a different directory then you’ll need to change these paths to match).
- Click on “C/C++ General” -> “Preprocessor Include Paths, Macros, etc.” property page:

- Click the “Providers” tab
  - \* In the list of providers, click “CDT Cross GCC Built-in Compiler Settings”. Change “Command to get compiler specs” to `xtensa-esp32-elf-gcc ${FLAGS} -E -P -v -dD "${INPUTS}"`.
  - \* In the list of providers, click “CDT GCC Build Output Parser” and change the “Compiler command pattern” to `xtensa-esp32-elf-(gcc|g\+\+|c\+\+|cc|cpp|clang)`

Navigate to “C/C++ General” -> “Indexer” property page:

- Check “Enable project specific settings” to enable the rest of the settings on this page.
- Uncheck “Allow heuristic resolution of includes”. When this option is enabled Eclipse sometimes fails to find correct header directories.

## Building in Eclipse

Continue from *Building in Eclipse* for all platforms.

## Technical Details

### Of interest to Windows gurus or very curious parties, only.

Explanations of the technical reasons for some of these steps. You don’t need to know this to use esp-idf with Eclipse on Windows, but it may be helpful background knowledge if you plan to do dig into the Eclipse support:

- The `xtensa-esp32-elf-gcc` cross-compiler is *not* a Cygwin toolchain, even though we tell Eclipse that it is one. This is because `msys2` uses Cygwin and supports Unix-style paths (of the type `/c/blah` instead of `c:/blah` or `c:\\blah`). In particular, `xtensa-esp32-elf-gcc` reports to the Eclipse “built-in compiler settings” function that its built-in include directories are all under `/usr/`, which is a Unix/Cygwin-style path that Eclipse otherwise can’t resolve. By telling Eclipse the compiler is Cygwin, it resolves these paths internally using the `cygpath` utility.
- The same problem occurs when parsing make output from esp-idf. Eclipse parses this output to find header directories, but it can’t resolve include directories of the form `/c/blah` without using `cygpath`. There is a heuristic that Eclipse Build Output Parser uses to determine whether it should call `cygpath`, but for currently unknown reasons the esp-idf configuration doesn’t trigger it. For this reason, the `eclipse_make.py` wrapper script is used to call `make` and then use `cygpath` to process the output for Eclipse.

## 1.13.5 IDF Monitor

The `idf_monitor` tool is a Python program which runs when the `make monitor` target is invoked in IDF.

It is mainly a serial terminal program which relays serial data to and from the target device’s serial port, but it has some other IDF-specific xfeatures.

### Interacting With `idf_monitor`

- `Ctrl-]` will exit the monitor.
- `Ctrl-T Ctrl-H` will display a help menu with all other keyboard shortcuts.
- Any other key apart from `Ctrl-]` and `Ctrl-T` is sent through the serial port.

## Automatically Decoding Addresses

Any time esp-idf prints a hexadecimal code address of the form `0x4_____`, `idf_monitor` will use `addr2line` to look up the source code location and function name.

When an esp-idf app crashes and panics a register dump and backtrace such as this is produced:

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

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

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

Behind the scenes, the command `idf_monitor` runs to decode each address is:

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

## Launch GDB for GDBStub

By default, if an esp-idf app crashes then the panic handler prints registers and a stack dump as shown above, and then resets.

Optionally, the panic handler can be configured to run a serial “gdb stub” which can communicate with a [gdb](#) debugger program and allow memory to be read, variables and stack frames examined, etc. This is not as versatile as JTAG debugging, but no special hardware is required.

To enable the gdbstub, run `make menuconfig` and set `ESP32_PANIC` option to Invoke GDBStub.

If this option is enabled and `idf_monitor` sees the gdb stub has loaded, it will automatically pause serial monitoring and run GDB with the correct arguments. After GDB exits, the board will be reset via the RTS serial line (if this is connected.)

Behind the scenes, the command `idf_monitor` runs is:

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

## Quick Compile and Flash

The keyboard shortcut `Ctrl-T Ctrl-F` will pause `idf_monitor`, run the `make flash target`, then resume `idf_monitor`. Any changed source files will be recompiled before re-flashing.

The keyboard shortcut `Ctrl-T Ctrl-A` will pause `idf_monitor`, run the `make app-flash target`, then resume `idf_monitor`. This is similar to `make flash`, but only the main app is compiled and reflashed.

## Quick Reset

The keyboard shortcut `Ctrl-T Ctrl-R` will reset the target board via the RTS line (if it is connected.)

## Pause the Application

The keyboard shortcut `Ctrl-T Ctrl-P` will reset the target into bootloader, so that the board will run nothing. This is useful when you want to wait for another device to startup. Then shortcut `Ctrl-T Ctrl-R` can be used to restart the application.

## Toggle Output Display

Sometimes you may want to stop new output printed to screen, to see the log before. The keyboard shortcut `Ctrl-T Ctrl-Y` will toggle the display (discard all serial data when the display is off) so that you can stop to see the log, and revert again quickly without quitting the monitor.

## Simple Monitor

Earlier versions of ESP-IDF used the `pySerial` command line program `miniterm` as a serial console program.

This program can still be run, via `make simple_monitor`.

`idf_monitor` is based on `miniterm` and shares the same basic keyboard shortcuts.

## Known Issues with idf\_monitor

### Issues Observed on Windows

- If you are using the supported Windows environment and receive the error “winpty: command not found” then run `pacman -S winpty` to fix.
- Arrow keys and some other special keys in gdb don’t work, due to Windows Console limitations.
- Occasionally when “make” exits, it may stall for up to 30 seconds before idf\_monitor resumes.
- Occasionally when “gdb” is run, it may stall for a short time before it begins communicating with the gdbstub.

### 1.13.6 Customized Setup of Toolchain

Instead of downloading binary toolchain from Espressif website (see [Setup Toolchain](#)) you may build the toolchain yourself.

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

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

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



## 2.1 Wi-Fi API

### 2.1.1 Wi-Fi

#### Introduction

The WiFi libraries provide support for configuring and monitoring the ESP32 WiFi networking functionality. This includes configuration for:

- Station mode (aka STA mode or WiFi client mode). ESP32 connects to an access point.
- AP mode (aka Soft-AP mode or Access Point mode). Stations connect to the ESP32.
- Combined AP-STA mode (ESP32 is concurrently an access point and a station connected to another access point).
- Various security modes for the above (WPA, WPA2, WEP, etc.)
- Scanning for access points (active & passive scanning).
- Promiscuous mode monitoring of IEEE802.11 WiFi packets.

#### Application Examples

See [wifi](#) directory of ESP-IDF examples that contains the following applications:

- Simple application showing how to connect ESP32 module to an Access Point - [esp-idf-template](#).
- Using power save mode of Wi-Fi - [wifi/power\\_save](#).

#### API Reference

## Header File

- `esp32/include/esp_wifi.h`

## Functions

`esp_err_t esp_wifi_init (const wifi_init_config_t *config)`

Init WiFi Alloc resource for WiFi driver, such as WiFi control structure, RX/TX buffer, WiFi NVS structure etc, this WiFi also start 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 init the config to default values, this can guarantee all the fields got correct value when more fields are added into *wifi\_init\_config\_t* in future release. If you want to set your own initial values, overwrite the default values which are set by `WIFI_INIT_CONFIG_DEFAULT`, please be notified that the field 'magic' of *wifi\_init\_config\_t* should always be `WIFI_INIT_CONFIG_MAGIC`!

### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NO_MEM`: out of memory
- others: refer to error code `esp_err.h`

### Parameters

- `config`: pointer to WiFi init configuration structure; can point to a temporary variable.

`esp_err_t esp_wifi_deinit (void)`

Deinit WiFi Free all resource allocated in `esp_wifi_init` and stop WiFi task.

**Attention** 1. This API should be called if you want to remove WiFi driver from the system

**Return** `ESP_OK`: succeed

`esp_err_t esp_wifi_set_mode (wifi_mode_t mode)`

Set the WiFi operating mode.

Set the WiFi operating mode as station, soft-AP or station+soft-AP, The default mode is soft-AP mode.

### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_ARG`: invalid argument
- others: refer to error code in `esp_err.h`

### Parameters

- `mode`: WiFi operating mode

`esp_err_t esp_wifi_get_mode (wifi_mode_t *mode)`

Get current operating mode of WiFi.

### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- mode: store current WiFi mode

esp\_err\_t **esp\_wifi\_start** (void)

Start WiFi according to current configuration. If mode is WIFI\_MODE\_STA, it create station control block and start station. If mode is WIFI\_MODE\_AP, it create soft-AP control block and start soft-AP. If mode is WIFI\_MODE\_APSTA, it create soft-AP and station control block and start soft-AP and station.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument
- ESP\_ERR\_WIFI\_NO\_MEM: out of memory
- ESP\_ERR\_WIFI\_CONN: WiFi internal error, station or soft-AP control block wrong
- ESP\_ERR\_WIFI\_FAIL: other WiFi internal errors

esp\_err\_t **esp\_wifi\_stop** (void)

Stop WiFi. If mode is WIFI\_MODE\_STA, it stop station and free station control block. If mode is WIFI\_MODE\_AP, it stop soft-AP and free soft-AP control block. If mode is WIFI\_MODE\_APSTA, it stop station/soft-AP and free station/soft-AP control block.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init

esp\_err\_t **esp\_wifi\_restore** (void)

Restore WiFi stack persistent settings to default values.

This function will reset settings made using the following APIs:

- esp\_wifi\_get\_auto\_connect,
- esp\_wifi\_set\_protocol,
- esp\_wifi\_set\_config related
- esp\_wifi\_set\_mode

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init

esp\_err\_t **esp\_wifi\_connect** (void)

Connect the ESP32 WiFi station to the AP.

**Attention** 1. This API only impact WIFI\_MODE\_STA or WIFI\_MODE\_APSTA mode

**Attention** 2. If the ESP32 is connected to an AP, call `esp_wifi_disconnect` to disconnect.

**Attention** 3. The scanning triggered by `esp_wifi_start_scan()` will not be effective until connection between ESP32 and the AP is established. If ESP32 is scanning and connecting at the same time, ESP32 will abort scanning and return a warning message and error number `ESP_ERR_WIFI_STATE`. If you want to do reconnection after ESP32 received disconnect event, remember to add the maximum retry time, otherwise the called scan will not work. This is especially true when the AP doesn't exist, and you still try reconnection after ESP32 received disconnect event with the reason code `WIFI_REASON_NO_AP_FOUND`.

**Return**

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_START`: WiFi is not started by `esp_wifi_start`
- `ESP_ERR_WIFI_CONN`: WiFi internal error, station or soft-AP control block wrong
- `ESP_ERR_WIFI_SSID`: SSID of AP which station connects is invalid

`esp_err_t esp_wifi_disconnect` (void)

Disconnect the ESP32 WiFi station from the AP.

**Return**

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi was not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi was not started by `esp_wifi_start`
- `ESP_ERR_WIFI_FAIL`: other WiFi internal errors

`esp_err_t esp_wifi_clear_fast_connect` (void)

Currently this API is just an stub API.

**Return**

- `ESP_OK`: succeed
- others: fail

`esp_err_t esp_wifi_deauth_sta` (uint16\_t *aid*)

deauthenticate all stations or associated id equals to aid

**Return**

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_NOT_STARTED`: WiFi was not started by `esp_wifi_start`
- `ESP_ERR_WIFI_ARG`: invalid argument
- `ESP_ERR_WIFI_MODE`: WiFi mode is wrong

**Parameters**

- *aid*: when aid is 0, deauthenticate all stations, otherwise deauthenticate station whose associated id is aid

`esp_err_t esp_wifi_scan_start (const wifi_scan_config_t *config, bool block)`  
 Scan all available APs.

**Attention** If this API is called, the found APs are stored in WiFi driver dynamic allocated memory and the will be freed in `esp_wifi_scan_get_ap_records`, so generally, call `esp_wifi_scan_get_ap_records` to cause the memory to be freed once the scan is done

**Attention** The values of maximum active scan time and passive scan time per channel are limited to 1500 milliseconds. Values above 1500ms may cause station to disconnect from AP and are not recommended.

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by `esp_wifi_init`
- ESP\_ERR\_WIFI\_NOT\_STARTED: WiFi was not started by `esp_wifi_start`
- ESP\_ERR\_WIFI\_TIMEOUT: blocking scan is timeout
- ESP\_ERR\_WIFI\_STATE: wifi still connecting when invoke `esp_wifi_scan_start`
- others: refer to error code in `esp_err.h`

**Parameters**

- `config`: configuration of scanning
- `block`: if `block` is true, this API will block the caller until the scan is done, otherwise it will return immediately

`esp_err_t esp_wifi_scan_stop (void)`  
 Stop the scan in process.

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by `esp_wifi_init`
- ESP\_ERR\_WIFI\_NOT\_STARTED: WiFi is not started by `esp_wifi_start`

`esp_err_t esp_wifi_scan_get_ap_num (uint16_t *number)`  
 Get number of APs found in last scan.

**Attention** This API can only be called when the scan is completed, otherwise it may get wrong value.

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by `esp_wifi_init`
- ESP\_ERR\_WIFI\_NOT\_STARTED: WiFi is not started by `esp_wifi_start`
- ESP\_ERR\_WIFI\_ARG: invalid argument

**Parameters**

- `number`: store number of APIs found in last scan

`esp_err_t esp_wifi_scan_get_ap_records (uint16_t *number, wifi_ap_record_t *ap_records)`  
 Get AP list found in last scan.

### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_NOT\_STARTED: WiFi is not started by esp\_wifi\_start
- ESP\_ERR\_WIFI\_ARG: invalid argument
- ESP\_ERR\_WIFI\_NO\_MEM: out of memory

### Parameters

- number: As input param, it stores max AP number ap\_records can hold. As output param, it receives the actual AP number this API returns.
- ap\_records: *wifi\_ap\_record\_t* array to hold the found APs

esp\_err\_t **esp\_wifi\_sta\_get\_ap\_info** (*wifi\_ap\_record\_t* \*ap\_info)  
Get information of AP which the ESP32 station is associated with.

### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_CONN: The station interface don't initialized
- ESP\_ERR\_WIFI\_NOT\_CONNECT: The station is in disconnect status

### Parameters

- ap\_info: the *wifi\_ap\_record\_t* to hold AP information

esp\_err\_t **esp\_wifi\_set\_ps** (*wifi\_ps\_type\_t* type)  
Set current power save type.

**Attention** Default power save type is WIFI\_PS\_NONE.

**Return** ESP\_ERR\_WIFI\_NOT\_SUPPORT: not supported yet

### Parameters

- type: power save type

esp\_err\_t **esp\_wifi\_get\_ps** (*wifi\_ps\_type\_t* \*type)  
Get current power save type.

**Attention** Default power save type is WIFI\_PS\_NONE.

**Return** ESP\_ERR\_WIFI\_NOT\_SUPPORT: not supported yet

### Parameters

- type: store current power save type

esp\_err\_t **esp\_wifi\_set\_protocol** (*wifi\_interface\_t* ifx, uint8\_t protocol\_bitmap)  
Set protocol type of specified interface The default protocol is (WIFI\_PROTOCOL\_11BIWIFI\_PROTOCOL\_11GIWIFI\_PROTOCOL\_11GN)

**Attention** Currently we only support 802.11b or 802.11bg or 802.11bgn mode

### Return

- ESP\_OK: succeed

- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_IF: invalid interface
- others: refer to error codes in esp\_err.h

#### Parameters

- ifx: interfaces
- protocol\_bitmap: WiFi protocol bitmap

esp\_err\_t **esp\_wifi\_get\_protocol** (*wifi\_interface\_t ifx*, uint8\_t \**protocol\_bitmap*)  
Get the current protocol bitmap of the specified interface.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_IF: invalid interface
- ESP\_ERR\_WIFI\_ARG: invalid argument
- others: refer to error codes in esp\_err.h

#### Parameters

- ifx: interface
- protocol\_bitmap: store current WiFi protocol bitmap of interface ifx

esp\_err\_t **esp\_wifi\_set\_bandwidth** (*wifi\_interface\_t ifx*, *wifi\_bandwidth\_t bw*)  
Set the bandwidth of ESP32 specified interface.

**Attention** 1. API return false if try to configure an interface that is not enabled

**Attention** 2. WIFI\_BW\_HT40 is supported only when the interface support 11N

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_IF: invalid interface
- ESP\_ERR\_WIFI\_ARG: invalid argument
- others: refer to error codes in esp\_err.h

#### Parameters

- ifx: interface to be configured
- bw: bandwidth

esp\_err\_t **esp\_wifi\_get\_bandwidth** (*wifi\_interface\_t ifx*, *wifi\_bandwidth\_t \*bw*)  
Get the bandwidth of ESP32 specified interface.

**Attention** 1. API return false if try to get a interface that is not enable

#### Return

- ESP\_OK: succeed

- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_IF: invalid interface
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- ifx: interface to be configured
- bw: store bandwidth of interface ifx

esp\_err\_t **esp\_wifi\_set\_channel** (uint8\_t *primary*, *wifi\_second\_chan\_t* *second*)  
Set primary/secondary channel of ESP32.

**Attention** 1. This is a special API for sniffer

**Attention** 2. This API should be called after esp\_wifi\_start() or esp\_wifi\_set\_promiscuous()

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_IF: invalid interface
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- primary: for HT20, primary is the channel number, for HT40, primary is the primary channel
- second: for HT20, second is ignored, for HT40, second is the second channel

esp\_err\_t **esp\_wifi\_get\_channel** (uint8\_t \**primary*, *wifi\_second\_chan\_t* \**second*)  
Get the primary/secondary channel of ESP32.

**Attention** 1. API return false if try to get a interface that is not enable

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- primary: store current primary channel
- second: store current second channel

esp\_err\_t **esp\_wifi\_set\_country** (const *wifi\_country\_t* \**country*)  
configure country info

**Attention** 1. The default country is {.cc="CN", .schan=1, .nchan=13, policy=WIFI\_COUNTRY\_POLICY\_AUTO}

**Attention** 2. When the country policy is WIFI\_COUNTRY\_POLICY\_AUTO, the country info of the AP to which the station is connected is used. E.g. if the configured country info is {.cc="USA", .schan=1, .nchan=11} and the country info of the AP to which the station is connected is {.cc="JP", .schan=1, .nchan=14} then the country info that will be used is {.cc="JP", .schan=1, .nchan=14}. If the station

disconnected from the AP the country info is set back to the country info of the station automatically, {.cc="USA", .schan=1, .nchan=11} in the example.

**Attention** 3. When the country policy is WIFI\_COUNTRY\_POLICY\_MANUAL, always use the configured country info.

**Attention** 4. When the country info is changed because of configuration or because the station connects to a different external AP, the country IE in probe response/beacon of the soft-AP is changed also.

**Attention** 5. The country configuration is not stored into flash

**Attention** 6. This API doesn't validate the per-country rules, it's up to the user to fill in all fields according to local regulations.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- `country`: the configured country info

esp\_err\_t **esp\_wifi\_get\_country** (*wifi\_country\_t* \*country)  
get the current country info

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- `country`: country info

esp\_err\_t **esp\_wifi\_set\_mac** (*wifi\_interface\_t* ifx, const uint8\_t mac[6])  
Set MAC address of the ESP32 WiFi station or the soft-AP interface.

**Attention** 1. This API can only be called when the interface is disabled

**Attention** 2. ESP32 soft-AP and station have different MAC addresses, do not set them to be the same.

**Attention** 3. The bit 0 of the first byte of ESP32 MAC address can not be 1. For example, the MAC address can set to be "1a:XX:XX:XX:XX:XX", but can not be "15:XX:XX:XX:XX:XX".

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument
- ESP\_ERR\_WIFI\_IF: invalid interface
- ESP\_ERR\_WIFI\_MAC: invalid mac address
- ESP\_ERR\_WIFI\_MODE: WiFi mode is wrong
- others: refer to error codes in esp\_err.h

#### Parameters

- `ifx`: interface
- `mac`: the MAC address

`esp_err_t esp_wifi_get_mac(wifi_interface_t ifx, uint8_t mac[6])`

Get mac of specified interface.

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_ARG`: invalid argument
- `ESP_ERR_WIFI_IF`: invalid interface

#### Parameters

- `ifx`: interface
- `mac`: store mac of the interface `ifx`

`esp_err_t esp_wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)`

Register the RX callback function in the promiscuous mode.

Each time a packet is received, the registered callback function will be called.

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`

#### Parameters

- `cb`: callback

`esp_err_t esp_wifi_set_promiscuous(bool en)`

Enable the promiscuous mode.

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`

#### Parameters

- `en`: false - disable, true - enable

`esp_err_t esp_wifi_get_promiscuous(bool *en)`

Get the promiscuous mode.

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_ARG`: invalid argument

#### Parameters

- `en`: store the current status of promiscuous mode

`esp_err_t esp_wifi_set_promiscuous_filter(const wifi_promiscuous_filter_t *filter)`  
Enable the promiscuous mode packet type filter.

**Note** The default filter is to filter all packets except `WIFI_PKT_MISC`

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`

#### Parameters

- `filter`: the packet type filtered in promiscuous mode.

`esp_err_t esp_wifi_get_promiscuous_filter(wifi_promiscuous_filter_t *filter)`  
Get the promiscuous filter.

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_ARG`: invalid argument

#### Parameters

- `filter`: store the current status of promiscuous filter

`esp_err_t esp_wifi_set_config(wifi_interface_t interface, wifi_config_t *conf)`  
Set the configuration of the ESP32 STA or AP.

**Attention** 1. This API can be called only when specified interface is enabled, otherwise, API fail

**Attention** 2. For station configuration, `bssid_set` needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

**Attention** 3. ESP32 is limited to only one channel, so when in the soft-AP+station mode, the soft-AP will adjust its channel automatically to be the same as the channel of the ESP32 station.

#### Return

- `ESP_OK`: succeed
- `ESP_ERR_WIFI_NOT_INIT`: WiFi is not initialized by `esp_wifi_init`
- `ESP_ERR_WIFI_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`

#### Parameters

- `interface`: interface
- `conf`: station or soft-AP configuration

`esp_err_t esp_wifi_get_config(wifi_interface_t interface, wifi_config_t *conf)`  
Get configuration of specified interface.

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument
- ESP\_ERR\_WIFI\_IF: invalid interface

**Parameters**

- interface: interface
- conf: station or soft-AP configuration

`esp_err_t esp_wifi_ap_get_sta_list(wifi_sta_list_t *sta)`  
Get STAs associated with soft-AP.

**Attention** SSC only API

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument
- ESP\_ERR\_WIFI\_MODE: WiFi mode is wrong
- ESP\_ERR\_WIFI\_CONN: WiFi internal error, the station/soft-AP control block is invalid

**Parameters**

- sta: station list

`esp_err_t esp_wifi_set_storage(wifi_storage_t storage)`  
Set the WiFi API configuration storage type.

**Attention** 1. The default value is WIFI\_STORAGE\_FLASH

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument

**Parameters**

- storage: : storage type

`esp_err_t esp_wifi_set_auto_connect(bool en)`  
Set auto connect The default value is true.

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init

- ESP\_ERR\_WIFI\_MODE: WiFi internal error, the station/soft-AP control block is invalid
- others: refer to error code in esp\_err.h

#### Parameters

- en: : true - enable auto connect / false - disable auto connect

esp\_err\_t **esp\_wifi\_get\_auto\_connect** (bool \*en)

Get the auto connect flag.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- en: store current auto connect configuration

esp\_err\_t **esp\_wifi\_set\_vendor\_ie** (bool enable, wifi\_vendor\_ie\_type\_t type, wifi\_vendor\_ie\_id\_t idx, const void \*vnd\_ie)

Set 802.11 Vendor-Specific Information Element.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init()
- ESP\_ERR\_WIFI\_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\_WIFI\_NO\_MEM: Out of memory

#### Parameters

- enable: If true, specified IE is enabled. If false, specified IE is removed.
- type: Information Element type. Determines the frame type to associate with the IE.
- idx: Index to set or clear. Each IE type can be associated with up to two elements (indices 0 & 1).
- vnd\_ie: Pointer to vendor specific element data. First 6 bytes should be a header with fields matching *wifi\_vendor\_ie\_data\_t*. If enable is false, this argument is ignored and can be NULL. Data does not need to remain valid after the function returns.

esp\_err\_t **esp\_wifi\_set\_vendor\_ie\_cb** (esp\_vendor\_ie\_cb\_t cb, void \*ctx)

Register Vendor-Specific Information Element monitoring callback.

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init

#### Parameters

- cb: Callback function
- ctx: Context argument, passed to callback function.

esp\_err\_t **esp\_wifi\_set\_max\_tx\_power** (int8\_t *power*)

Set maximum WiFi transmitting power.

**Attention** WiFi transmitting power is divided to six levels in phy init data. Level0 represents highest transmitting power and level5 represents lowest transmitting power. Packets of different rates are transmitted in different powers according to the configuration in phy init data. This API only sets maximum WiFi transmitting power. If this API is called, the transmitting power of every packet will be less than or equal to the value set by this API. If this API is not called, the value of maximum transmitting power set in phy\_init\_data.bin or menuconfig (depend on whether to use phy init data in partition or not) will be used. Default value is level0. Values passed in power are mapped to transmit power levels as follows:

- [78, 127]: level0
- [76, 77]: level1
- [74, 75]: level2
- [68, 73]: level3
- [60, 67]: level4
- [52, 59]: level5
- [44, 51]: level5 - 2dBm
- [34, 43]: level5 - 4.5dBm
- [28, 33]: level5 - 6dBm
- [20, 27]: level5 - 8dBm
- [8, 19]: level5 - 11dBm
- [-128, 7]: level5 - 14dBm

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_NOT\_START: WiFi is not started by esp\_wifi\_start

#### Parameters

- *power*: Maximum WiFi transmitting power.

esp\_err\_t **esp\_wifi\_get\_max\_tx\_power** (int8\_t \**power*)

Get maximum WiFi transmitting power.

**Attention** This API gets maximum WiFi transmitting power. Values got from power are mapped to transmit power levels as follows:

- 78: 19.5dBm
- 76: 19dBm
- 74: 18.5dBm
- 68: 17dBm
- 60: 15dBm
- 52: 13dBm
- 44: 11dBm

- 34: 8.5dBm
- 28: 7dBm
- 20: 5dBm
- 8: 2dBm
- -4: -1dBm

#### Return

- ESP\_OK: succeed
- ESP\_ERR\_WIFI\_NOT\_INIT: WiFi is not initialized by esp\_wifi\_init
- ESP\_ERR\_WIFI\_NOT\_START: WiFi is not started by esp\_wifi\_start
- ESP\_ERR\_WIFI\_ARG: invalid argument

#### Parameters

- power: Maximum WiFi transmitting power.

## Structures

### struct wifi\_init\_config\_t

WiFi stack configuration parameters passed to esp\_wifi\_init call.

#### Public Members

system\_event\_handler\_t **event\_handler**

WiFi event handler

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 **ampdu\_rx\_enable**

WiFi AMPDU RX feature enable flag

int **ampdu\_tx\_enable**

WiFi AMPDU TX feature enable flag

int **nvs\_enable**

WiFi NVS flash enable flag

int **nano\_enable**

Nano option for printf/scan family enable flag

int **tx\_ba\_win**  
WiFi Block Ack TX window size

int **rx\_ba\_win**  
WiFi Block Ack RX window size

int **magic**  
WiFi init magic number, it should be the last field

## Macros

**ESP\_ERR\_WIFI\_OK**  
No error

**ESP\_ERR\_WIFI\_FAIL**  
General fail code

**ESP\_ERR\_WIFI\_NO\_MEM**  
Out of memory

**ESP\_ERR\_WIFI\_ARG**  
Invalid argument

**ESP\_ERR\_WIFI\_NOT\_SUPPORT**  
Indicates that API is not supported yet

**ESP\_ERR\_WIFI\_NOT\_INIT**  
WiFi driver was not installed by esp\_wifi\_init

**ESP\_ERR\_WIFI\_NOT\_STARTED**  
WiFi driver was not started by esp\_wifi\_start

**ESP\_ERR\_WIFI\_NOT\_STOPPED**  
WiFi driver was not stopped by esp\_wifi\_stop

**ESP\_ERR\_WIFI\_IF**  
WiFi interface error

**ESP\_ERR\_WIFI\_MODE**  
WiFi mode error

**ESP\_ERR\_WIFI\_STATE**  
WiFi internal state error

**ESP\_ERR\_WIFI\_CONN**  
WiFi internal control block of station or soft-AP error

**ESP\_ERR\_WIFI\_NVS**  
WiFi internal NVS module error

**ESP\_ERR\_WIFI\_MAC**  
MAC address is invalid

**ESP\_ERR\_WIFI\_SSID**  
SSID is invalid

**ESP\_ERR\_WIFI\_PASSWORD**  
Password is invalid

**ESP\_ERR\_WIFI\_TIMEOUT**  
Timeout error

**ESP\_ERR\_WIFI\_WAKE\_FAIL**

WiFi is in sleep state(RF closed) and wakeup fail

**ESP\_ERR\_WIFI\_WOULD\_BLOCK**

The caller would block

**ESP\_ERR\_WIFI\_NOT\_CONNECT**

Station still in disconnect status

**WIFI\_STATIC\_TX\_BUFFER\_NUM****WIFI\_DYNAMIC\_TX\_BUFFER\_NUM****WIFI\_AMPDU\_RX\_ENABLED****WIFI\_AMPDU\_TX\_ENABLED****WIFI\_NVS\_ENABLED****WIFI\_NANO\_FORMAT\_ENABLED****WIFI\_INIT\_CONFIG\_MAGIC****WIFI\_DEFAULT\_TX\_BA\_WIN****WIFI\_DEFAULT\_RX\_BA\_WIN****WIFI\_INIT\_CONFIG\_DEFAULT()**

## Type Definitions

```
typedef void (*wifi_promiscuous_cb_t) (void *buf, wifi_promiscuous_pkt_type_t type)
```

The RX callback function in the promiscuous mode. Each time a packet is received, the callback function will be called.

### Parameters

- `buf`: Data received. Type of data in buffer (*wifi\_promiscuous\_pkt\_t* or *wifi\_pkt\_rx\_ctrl\_t*) indicated by 'type' parameter.
- `type`: promiscuous packet type.

```
typedef void (*esp_vendor_ie_cb_t) (void *ctx, wifi_vendor_ie_type_t type, const uint8_t sa[6],  
                                     const vendor_ie_data_t *vnd_ie, int rssi)
```

Function signature for received Vendor-Specific Information Element callback.

### Parameters

- `ctx`: Context argument, as passed to `esp_wifi_set_vendor_ie_cb()` when registering callback.
- `type`: Information element type, based on frame type received.
- `sa`: Source 802.11 address.
- `vnd_ie`: Pointer to the vendor specific element data received.
- `rssi`: Received signal strength indication.

## Header File

- `esp32/include/esp_wifi_types.h`

## Unions

### **union wifi\_scan\_time\_t**

*#include <esp\_wifi\_types.h>* 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.

### **union wifi\_config\_t**

*#include <esp\_wifi\_types.h>* Configuration data for ESP32 AP or STA.

The usage of this union (for ap or sta configuration) is determined by the accompanying interface argument passed to `esp_wifi_set_config()` or `esp_wifi_get_config()`

#### **Public Members**

##### *wifi\_ap\_config\_t* **ap**

configuration of AP

##### *wifi\_sta\_config\_t* **sta**

configuration of STA

## Structures

### **struct wifi\_country\_t**

Structure describing WiFi country-based regional restrictions.

#### **Public Members**

char **cc**[3]

country code string

uint8\_t **schan**

start channel

uint8\_t **nchan**

total channel number

*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\_config\_t**  
Parameters for an SSID scan.

## Public Members

`uint8_t *ssid`  
SSID of AP

`uint8_t *bssid`  
MAC address of AP

`uint8_t channel`  
channel, scan the specific channel

`bool show_hidden`  
enable to scan AP whose SSID is hidden

`wifi_scan_type_t scan_type`  
scan type, active or passive

`wifi_scan_time_t scan_time`  
scan time per channel

**struct wifi\_ap\_record\_t**  
Description of an WiFi AP.

## Public Members

`uint8_t bssid[6]`  
MAC address of AP

`uint8_t ssid[33]`  
SSID of AP

`uint8_t primary`  
channel of AP

`wifi_second_chan_t second`  
second channel of AP

`int8_t rssi`  
signal strength of AP

`wifi_auth_mode_t authmode`  
authmode of AP

`wifi_cipher_type_t pairwise_cipher`  
pairwise cipher of AP

`wifi_cipher_type_t group_cipher`  
group cipher of AP

`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_1r`  
bit: 3 flag to identify if low rate is enabled or not

`uint32_t wps`  
bit: 4 flag to identify if WPS is supported or not

`uint32_t reserved`  
bit: 5..31 reserved

**struct wifi\_fast\_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

**struct wifi\_ap\_config\_t**  
Soft-AP configuration settings for the ESP32.

### Public Members

`uint8_t ssid[32]`  
SSID of ESP32 soft-AP

`uint8_t password[64]`  
Password of ESP32 soft-AP

`uint8_t ssid_len`  
Length of SSID. If `softap_config.ssid_len==0`, check the SSID until there is a termination character; otherwise, set the SSID length according to `softap_config.ssid_len`.

`uint8_t channel`  
Channel of ESP32 soft-AP

*wifi\_auth\_mode\_t* `authmode`  
Auth mode of ESP32 soft-AP. Do not support AUTH\_WEP in soft-AP mode

`uint8_t ssid_hidden`  
Broadcast SSID or not, default 0, broadcast the SSID

`uint8_t max_connection`  
Max number of stations allowed to connect in, default 4, max 4

`uint16_t beacon_interval`  
Beacon interval, 100 ~ 60000 ms, default 100 ms

**struct wifi\_sta\_config\_t**  
STA configuration settings for the ESP32.

## Public Members

uint8\_t **ssid**[32]

SSID of target AP

uint8\_t **password**[64]

password of target AP

*wifi\_scan\_method\_t* **scan\_method**

do all channel scan or fast scan

bool **bssid\_set**

whether set MAC address of target AP or not. Generally, station\_config.bssid\_set needs to be 0; and it needs to be 1 only when users need to check the MAC address of the AP.

uint8\_t **bssid**[6]

MAC address of target AP

uint8\_t **channel**

channel of target AP. Set to 1~13 to scan starting from the specified channel before connecting to AP. If the channel of AP is unknown, set it to 0.

*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\_method is set, only APs which have an auth mode that is more secure than the selected auth mode and a signal stronger than the minimum RSSI will be used.

**struct wifi\_sta\_info\_t**

Description of STA associated with AP.

## Public Members

uint8\_t **mac**[6]

mac address

**struct wifi\_sta\_list\_t**

List of stations associated with the ESP32 Soft-AP.

## Public Members

*wifi\_sta\_info\_t* **sta**[ESP\_WIFI\_MAX\_CONN\_NUM]

station list

int **num**

number of stations in the list (other entries are invalid)

**struct vendor\_ie\_data\_t**

Vendor Information Element header.

The first bytes of the Information Element will match this header. Payload follows.

## Public Members

uint8\_t **element\_id**

Should be set to WIFI\_VENDOR\_IE\_ELEMENT\_ID (0xDD)

uint8\_t **length**

Length of all bytes in the element data following this field. Minimum 4.

uint8\_t **vendor\_oui**[3]

Vendor identifier (OUI).

uint8\_t **vendor\_oui\_type**

Vendor-specific OUI type.

uint8\_t **payload**[0]

Payload. Length is equal to value in 'length' field, minus 4.

**struct wifi\_pkt\_rx\_ctrl\_t**

Received packet radio metadata header, this is the common header at the beginning of all promiscuous mode RX callback buffers.

### Public Members

signed **rss\_i**

signal intensity of packet

unsigned **rate**

data rate

unsigned **\_\_pad0\_\_**

reserve

unsigned **sig\_mode**

0:is not 11n packet; 1:is 11n packet

unsigned **\_\_pad1\_\_**

reserve

unsigned **mcs**

if is 11n packet, shows the modulation(range from 0 to 76)

unsigned **cwb**

if is 11n packet, shows if is HT40 packet or not

unsigned **\_\_pad2\_\_**

reserve

unsigned **smoothing**

reserve

unsigned **not\_sounding**

reserve

unsigned **\_\_pad3\_\_**

reserve

unsigned **aggregation**

Aggregation

unsigned **stbc**

STBC

unsigned **fec\_coding**

Flag is set for 11n packets which are LDPC

unsigned **sgi**

SGI

unsigned **noise\_floor**  
noise floor

unsigned **ampdu\_cnt**  
ampdu cnt

unsigned **channel**  
which channel this packet in

unsigned **\_\_pad4\_\_**  
reserve

unsigned **timestamp**  
timestamp

unsigned **\_\_pad5\_\_**  
reserve

unsigned **\_\_pad6\_\_**  
reserve

unsigned **sig\_len**  
length of packet

unsigned **\_\_pad7\_\_**  
reserve

unsigned **rx\_state**  
rx state

**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\_\*

### Macros

**WIFI\_IF\_STA**

**WIFI\_IF\_AP**

**WIFI\_PROTOCOL\_11B**

**WIFI\_PROTOCOL\_11G**

**WIFI\_PROTOCOL\_11N**

**WIFI\_PROTOCOL\_LR**

**ESP\_WIFI\_MAX\_CONN\_NUM**

max number of stations which can connect to ESP32 soft-AP

**WIFI\_VENDOR\_IE\_ELEMENT\_ID**

**WIFI\_PROMIS\_FILTER\_MASK\_ALL**

filter all packets

**WIFI\_PROMIS\_FILTER\_MASK\_MGMT**

filter the packets with type of WIFI\_PKT\_MGMT

**WIFI\_PROMIS\_FILTER\_MASK\_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

## Type Definitions

```
typedef esp_interface_t wifi_interface_t
```

```
typedef wifi_fast_scan_threshold_t wifi_scan_threshold_t
```

*wifi\_fast\_scan\_threshold\_t* only used in fast scan mode once, now it enabled in all channel scan, the *wifi\_fast\_scan\_threshold\_t* will be remove in version 4.0

## Enumerations

```
enum wifi_mode_t
```

*Values:*

**WIFI\_MODE\_NULL** = 0

null mode

**WIFI\_MODE\_STA**

WiFi station mode

**WIFI\_MODE\_AP**

WiFi soft-AP mode

**WIFI\_MODE\_APSTA**

WiFi station + soft-AP mode

**WIFI\_MODE\_MAX**

```
enum wifi_country_policy_t
```

*Values:*

**WIFI\_COUNTRY\_POLICY\_AUTO**

Country policy is auto, use the country info of AP to which the station is connected

**WIFI\_COUNTRY\_POLICY\_MANUAL**

Country policy is manual, always use the configured country info

**enum wifi\_auth\_mode\_t**

*Values:*

**WIFI\_AUTH\_OPEN** = 0

authenticate mode : open

**WIFI\_AUTH\_WEP**

authenticate mode : WEP

**WIFI\_AUTH\_WPA\_PSK**

authenticate mode : WPA\_PSK

**WIFI\_AUTH\_WPA2\_PSK**

authenticate mode : WPA2\_PSK

**WIFI\_AUTH\_WPA\_WPA2\_PSK**

authenticate mode : WPA\_WPA2\_PSK

**WIFI\_AUTH\_WPA2\_ENTERPRISE**

authenticate mode : WPA2\_ENTERPRISE

**WIFI\_AUTH\_MAX**

**enum wifi\_err\_reason\_t**

*Values:*

**WIFI\_REASON\_UNSPECIFIED** = 1

**WIFI\_REASON\_AUTH\_EXPIRE** = 2

**WIFI\_REASON\_AUTH\_LEAVE** = 3

**WIFI\_REASON\_ASSOC\_EXPIRE** = 4

**WIFI\_REASON\_ASSOC\_TOOMANY** = 5

**WIFI\_REASON\_NOT\_AUTHED** = 6

**WIFI\_REASON\_NOT\_ASSOCED** = 7

**WIFI\_REASON\_ASSOC\_LEAVE** = 8

**WIFI\_REASON\_ASSOC\_NOT\_AUTHED** = 9

**WIFI\_REASON\_DISASSOC\_PWRCAP\_BAD** = 10

**WIFI\_REASON\_DISASSOC\_SUPCHAN\_BAD** = 11

**WIFI\_REASON\_IE\_INVALID** = 13

**WIFI\_REASON\_MIC\_FAILURE** = 14

**WIFI\_REASON\_4WAY\_HANDSHAKE\_TIMEOUT** = 15

**WIFI\_REASON\_GROUP\_KEY\_UPDATE\_TIMEOUT** = 16

**WIFI\_REASON\_IE\_IN\_4WAY\_DIFFERS** = 17

**WIFI\_REASON\_GROUP\_CIPHER\_INVALID** = 18

**WIFI\_REASON\_PAIRWISE\_CIPHER\_INVALID** = 19

**WIFI\_REASON\_AKMP\_INVALID** = 20

**WIFI\_REASON\_UNSUPP\_RSN\_IE\_VERSION** = 21

**WIFI\_REASON\_INVALID\_RSN\_IE\_CAP** = 22  
**WIFI\_REASON\_802\_1X\_AUTH\_FAILED** = 23  
**WIFI\_REASON\_CIPHER\_SUITE\_REJECTED** = 24  
**WIFI\_REASON\_BEACON\_TIMEOUT** = 200  
**WIFI\_REASON\_NO\_AP\_FOUND** = 201  
**WIFI\_REASON\_AUTH\_FAIL** = 202  
**WIFI\_REASON\_ASSOC\_FAIL** = 203  
**WIFI\_REASON\_HANDSHAKE\_TIMEOUT** = 204  
**WIFI\_REASON\_CONNECTION\_FAIL** = 205

**enum wifi\_second\_chan\_t**

*Values:*

**WIFI\_SECOND\_CHAN\_NONE** = 0  
the channel width is HT20  
**WIFI\_SECOND\_CHAN\_ABOVE**  
the channel width is HT40 and the second channel is above the primary channel  
**WIFI\_SECOND\_CHAN\_BELOW**  
the channel width is HT40 and the second channel is below the primary channel

**enum wifi\_scan\_type\_t**

*Values:*

**WIFI\_SCAN\_TYPE\_ACTIVE** = 0  
active scan  
**WIFI\_SCAN\_TYPE\_PASSIVE**  
passive scan

**enum wifi\_cipher\_type\_t**

*Values:*

**WIFI\_CIPHER\_TYPE\_NONE** = 0  
the cipher type is none  
**WIFI\_CIPHER\_TYPE\_WEP40**  
the cipher type is WEP40  
**WIFI\_CIPHER\_TYPE\_WEP104**  
the cipher type is WEP104  
**WIFI\_CIPHER\_TYPE\_TKIP**  
the cipher type is TKIP  
**WIFI\_CIPHER\_TYPE\_CCMP**  
the cipher type is CCMP  
**WIFI\_CIPHER\_TYPE\_TKIP\_CCMP**  
the cipher type is TKIP and CCMP  
**WIFI\_CIPHER\_TYPE\_UNKNOWN**  
the cipher type is unknown

**enum wifi\_scan\_method\_t**

*Values:*

**WIFI\_FAST\_SCAN** = 0

Do fast scan, scan will end after find SSID match AP

**WIFI\_ALL\_CHANNEL\_SCAN**

All channel scan, scan will end after scan all the channel

**enum wifi\_sort\_method\_t**

*Values:*

**WIFI\_CONNECT\_AP\_BY\_SIGNAL** = 0

Sort match AP in scan list by RSSI

**WIFI\_CONNECT\_AP\_BY\_SECURITY**

Sort match AP in scan list by security mode

**enum wifi\_ps\_type\_t**

*Values:*

**WIFI\_PS\_NONE**

No power save

**WIFI\_PS\_MODEM**

Modem power save

**enum wifi\_bandwidth\_t**

*Values:*

**WIFI\_BW\_HT20** = 1

**WIFI\_BW\_HT40**

**enum wifi\_storage\_t**

*Values:*

**WIFI\_STORAGE\_FLASH**

all configuration will store in both memory and flash

**WIFI\_STORAGE\_RAM**

all configuration will only store in the memory

**enum wifi\_vendor\_ie\_type\_t**

Vendor Information Element type.

Determines the frame type that the IE will be associated with.

*Values:*

**WIFI\_VND\_IE\_TYPE\_BEACON**

**WIFI\_VND\_IE\_TYPE\_PROBE\_REQ**

**WIFI\_VND\_IE\_TYPE\_PROBE\_RESP**

**WIFI\_VND\_IE\_TYPE\_ASSOC\_REQ**

**WIFI\_VND\_IE\_TYPE\_ASSOC\_RESP**

**enum wifi\_vendor\_ie\_id\_t**

Vendor Information Element index.

Each IE type can have up to two associated vendor ID elements.

*Values:*

**WIFI\_VND\_IE\_ID\_0**

**WIFI\_VND\_IE\_ID\_1**

**enum wifi\_promiscuous\_pkt\_type\_t**

Promiscuous frame type.

Passed to promiscuous mode RX callback to indicate the type of parameter in the buffer.

*Values:*

**WIFI\_PKT\_MGMT**

Management frame, indicates 'buf' argument is *wifi\_promiscuous\_pkt\_t*

**WIFI\_PKT\_DATA**

Data frame, indicates 'buf' argument is *wifi\_promiscuous\_pkt\_t*

**WIFI\_PKT\_MISC**

Other type, such as MIMO etc. 'buf' argument is *wifi\_promiscuous\_pkt\_t* but the payload is zero length.

## 2.1.2 Smart Config

### API Reference

#### Header File

- [esp32/include/esp\\_smartconfig.h](#)

#### Functions

**const char \***`esp_smartconfig_get_version` (void)

Get the version of SmartConfig.

#### Return

- SmartConfig version const char.

`esp_err_t` **esp\_smartconfig\_start** (*sc\_callback\_t* cb, ...)

Start SmartConfig, config ESP device to connect AP. You need to broadcast information by phone APP. Device sniffer special packets from the air that containing SSID and password of target AP.

**Attention** 1. This API can be called in station or softAP-station mode.

**Attention** 2. Can not call `esp_smartconfig_start` twice before it finish, please call `esp_smartconfig_stop` first.

#### Return

- ESP\_OK: succeed
- others: fail

#### Parameters

- cb: SmartConfig callback function.
- . . . : log 1: UART output logs; 0: UART only outputs the result.

`esp_err_t` **esp\_smartconfig\_stop** (void)

Stop SmartConfig, free the buffer taken by `esp_smartconfig_start`.

**Attention** Whether connect to AP succeed or not, this API should be called to free memory taken by `smartconfig_start`.

**Return**

- ESP\_OK: succeed
- others: fail

esp\_err\_t **esp\_esptouch\_set\_timeout** (uint8\_t *time\_s*)

Set timeout of SmartConfig process.

**Attention** Timing starts from SC\_STATUS\_FIND\_CHANNEL status. SmartConfig will restart if timeout.

**Return**

- ESP\_OK: succeed
- others: fail

**Parameters**

- *time\_s*: range 15s~255s, offset:45s.

esp\_err\_t **esp\_smartconfig\_set\_type** (*smartconfig\_type\_t type*)

Set protocol type of SmartConfig.

**Attention** If users need to set the SmartConfig type, please set it before calling esp\_smartconfig\_start.

**Return**

- ESP\_OK: succeed
- others: fail

**Parameters**

- *type*: Choose from the smartconfig\_type\_t.

esp\_err\_t **esp\_smartconfig\_fast\_mode** (bool *enable*)

Set mode of SmartConfig. default normal mode.

**Attention** 1. Please call it before API esp\_smartconfig\_start.

**Attention** 2. Fast mode have corresponding APP(phone).

**Attention** 3. Two mode is compatible.

**Return**

- ESP\_OK: succeed
- others: fail

**Parameters**

- *enable*: false-disable(default); true-enable;

**Type Definitions**

**typedef** void (\***sc\_callback\_t**) (*smartconfig\_status\_t status*, void \*pdata)

The callback of SmartConfig, executed when smart-config status changed.

**Parameters**

- *status*: Status of SmartConfig;

- SC\_STATUS\_GETTING\_SSID\_PSWD : pdata is a pointer of smartconfig\_type\_t, means config type.
  - SC\_STATUS\_LINK : pdata is a pointer of struct station\_config.
  - SC\_STATUS\_LINK\_OVER : pdata is a pointer of phone's IP address(4 bytes) if pdata unequal NULL.
  - otherwise : parameter void \*pdata is NULL.
- pdata: According to the different status have different values.

## Enumerations

### enum smartconfig\_status\_t

*Values:*

- SC\_STATUS\_WAIT** = 0  
Waiting to start connect
- SC\_STATUS\_FIND\_CHANNEL**  
Finding target channel
- SC\_STATUS\_GETTING\_SSID\_PSWD**  
Getting SSID and password of target AP
- SC\_STATUS\_LINK**  
Connecting to target AP
- SC\_STATUS\_LINK\_OVER**  
Connected to AP successfully

### enum smartconfig\_type\_t

*Values:*

- SC\_TYPE\_ESPTOUCH** = 0  
protocol: ESPTouch
- SC\_TYPE\_AIRKISS**  
protocol: AirKiss
- SC\_TYPE\_ESPTOUCH\_AIRKISS**  
protocol: ESPTouch and AirKiss

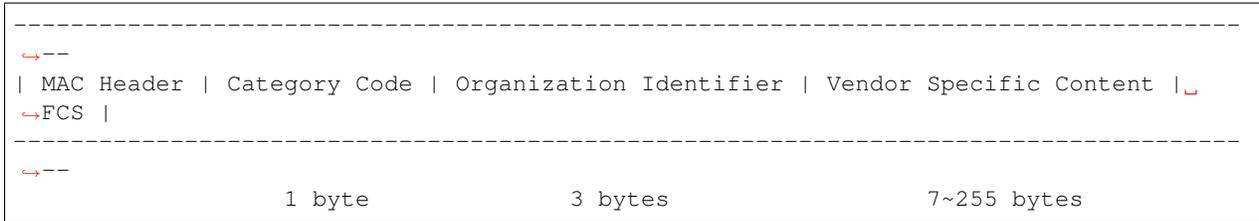
## 2.1.3 ESP-NOW

### Overview

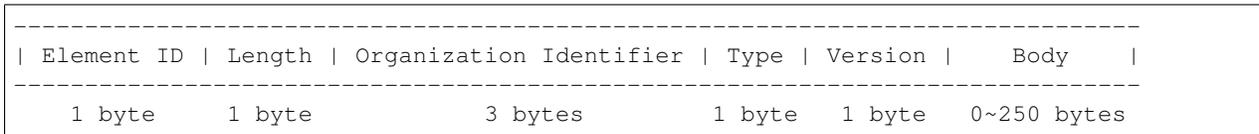
ESP-NOW is a kind of connectionless WiFi communication protocol which is defined by Espressif. In ESP-NOW, application data is encapsulated in vendor-specific action frame and then transmitted from one WiFi device to another without connection. CTR with CBC-MAC Protocol(CCMP) is used to protect the action frame for security. ESP-NOW is widely used in smart light, remote controlling, sensor, etc.

### Frame Format

ESP-NOW uses vendor-specific action frame to transmit ESP-NOW data. The format of vendor-specific action frame is as follows:



- Category Code: The Category 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.**
- Vendor Specific Content: The Vendor Specific Content contains vendor-specific field as follows:



- Element ID: The Element ID field is set to the value(221) indicating the vendor-specific element.
- Length: The length is the total length of Organization Identifier, Type, Version and Body.
- **Organization Identifier: The Organization Identifier contains a unique identifier(0x18fe34) which is the first three bytes of MAC address applied by Espressif.**
- Type: The Type field is set to the value(4) indicating ESP-NOW.
- Version: The Version field is set to the version of ESP-NOW.
- Body: The Body contains the ESP-NOW data.

As ESP-NOW is connectionless, the MAC header is a little different from that of standard frames. The FromDS and ToDS bits of FrameControl field are both 0. The first address field is set to the destination address. The second address field is set to the source address. The third address field is set to broadcast address(0xff:0xff:0xff:0xff:0xff:0xff).

## Security

ESP-NOW use CCMP method which can be referenced in IEEE Std. 802.11-2012 to protect the vendor-specific action frame. The WiFi device maintains a Primary Master Key(PMK) and several Local Master Keys(LMK). The lengths of them are 16 bytes. PMK is used to encrypt LMK with AES-128 algorithm. Call `esp_now_set_pmk()` to set PMK. If PMK is not set, a default PMK will be used. If LMK of the paired device is set, it will be used to encrypt the vendor-specific action frame with CCMP method. The maximum number of different LMKs is six. Do not support encrypting multicast vendor-specific action frame.

## Initialization and De-initialization

Call `esp_now_init()` to initialize ESP-NOW and `esp_now_deinit()` to de-initialize ESP-NOW. ESP-NOW data must be transmitted after WiFi is started, so it is recommended to start WiFi before initializing ESP-NOW and stop WiFi after de-initializing ESP-NOW. When `esp_now_deinit()` is called, all of the information of paired devices will be deleted.

## Add Paired Device

Before sending data to other device, call `esp_now_add_peer()` to add it to the paired device list first. The maximum number of paired devices is twenty. If security is enabled, the LMK must be set. ESP-NOW data can be

sent from station or softap interface. Make sure that the interface is enabled before sending ESP-NOW data. A device with broadcast MAC address must be added before sending broadcast data. The range of the channel of paired device 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 MAC layer. Otherwise, it will return `ESP_NOW_SEND_FAIL`. There are several reasons failing to send ESP-NOW data, for example, the destination device doesn't exist, the channels of the devices are not the same, the action frame is lost when transmitting on the air, etc. It is not guaranteed that application layer can receive the data. If necessary, send back ack data when receiving ESP-NOW data. If receiving ack data timeout happens, retransmit the ESP-NOW data. A sequence number can also be assigned to ESP-NOW data to drop the duplicated 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 datas 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 previous sending has returned. The sending callback function runs from a high-priority WiFi 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.

### Receiving ESP-NOW Data

Call `esp_now_register_recv_cb` to register receiving callback function. When receiving ESP-NOW data, receiving callback function is called. The receiving callback function also runs from WiFi 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.

## API Reference

### Header File

- `esp32/include/esp_now.h`

### Functions

`esp_err_t esp_now_init (void)`  
Initialize ESPNOW function.

#### Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_INTERNAL` : Internal error

`esp_err_t esp_now_deinit (void)`  
De-initialize ESPNOW function.

#### Return

- `ESP_OK` : succeed

`esp_err_t esp_now_get_version (uint32_t *version)`  
Get the version of ESPNOW.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_ARG : invalid argument

**Parameters**

- *version*: ESPNOW version

`esp_err_t esp_now_register_recv_cb(esp_now_recv_cb_t cb)`  
Register callback function of receiving ESPNOW data.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized
- ESP\_ERR\_ESPNOW\_INTERNAL : internal error

**Parameters**

- *cb*: callback function of receiving ESPNOW data

`esp_err_t esp_now_unregister_recv_cb(void)`  
Unregister callback function of receiving ESPNOW data.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized

`esp_err_t esp_now_register_send_cb(esp_now_send_cb_t cb)`  
Register callback function of sending ESPNOW data.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized
- ESP\_ERR\_ESPNOW\_INTERNAL : internal error

**Parameters**

- *cb*: callback function of sending ESPNOW data

`esp_err_t esp_now_unregister_send_cb(void)`  
Unregister callback function of sending ESPNOW data.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized

`esp_err_t esp_now_send(const uint8_t *peer_addr, const uint8_t *data, size_t len)`  
Send ESPNOW data.

**Attention** 1. If *peer\_addr* is not NULL, send data to the peer whose MAC address matches *peer\_addr*

**Attention** 2. If `peer_addr` is NULL, send data to all of the peers that are added to the peer list

**Attention** 3. The maximum length of data must be less than `ESP_NOW_MAX_DATA_LEN`

**Attention** 4. The buffer pointed to by data argument does not need to be valid after `esp_now_send` returns

#### Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_INTERNAL` : internal error
- `ESP_ERR_ESPNOW_NO_MEM` : out of memory
- `ESP_ERR_ESPNOW_NOT_FOUND` : peer is not found
- `ESP_ERR_ESPNOW_IF` : current WiFi interface doesn't match that of peer

#### Parameters

- `peer_addr`: peer MAC address
- `data`: data to send
- `len`: length of data

`esp_err_t esp_now_add_peer (const esp_now_peer_info_t *peer)`

Add a peer to peer list.

#### Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_FULL` : peer list is full
- `ESP_ERR_ESPNOW_NO_MEM` : out of memory
- `ESP_ERR_ESPNOW_EXIST` : peer has existed

#### Parameters

- `peer`: peer information

`esp_err_t esp_now_del_peer (const uint8_t *peer_addr)`

Delete a peer from peer list.

#### Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument
- `ESP_ERR_ESPNOW_NOT_FOUND` : peer is not found

#### Parameters

- `peer_addr`: peer MAC address

`esp_err_t esp_now_mod_peer (const esp_now_peer_info_t *peer)`  
Modify a peer.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized
- ESP\_ERR\_ESPNOW\_ARG : invalid argument
- ESP\_ERR\_ESPNOW\_FULL : peer list is full

**Parameters**

- peer: peer information

`esp_err_t esp_now_get_peer (const uint8_t *peer_addr, esp_now_peer_info_t *peer)`  
Get a peer whose MAC address matches peer\_addr from peer list.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized
- ESP\_ERR\_ESPNOW\_ARG : invalid argument
- ESP\_ERR\_ESPNOW\_NOT\_FOUND : peer is not found

**Parameters**

- peer\_addr: peer MAC address
- peer: peer information

`esp_err_t esp_now_fetch_peer (bool from_head, esp_now_peer_info_t *peer)`  
Fetch a peer from peer list.

**Return**

- ESP\_OK : succeed
- ESP\_ERR\_ESPNOW\_NOT\_INIT : ESPNOW is not initialized
- ESP\_ERR\_ESPNOW\_ARG : invalid argument
- ESP\_ERR\_ESPNOW\_NOT\_FOUND : peer is not found

**Parameters**

- from\_head: fetch from head of list or not
- peer: peer information

`bool esp_now_is_peer_exist (const uint8_t *peer_addr)`  
Peer exists or not.

**Return**

- true : peer exists
- false : peer not exists

**Parameters**

- `peer_addr`: peer MAC address

`esp_err_t esp_now_get_peer_num(esp_now_peer_num_t *num)`  
Get the number of peers.

#### Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument

#### Parameters

- `num`: number of peers

`esp_err_t esp_now_set_pmk(const uint8_t *pmk)`  
Set the primary master key.

**Attention** 1. primary master key is used to encrypt local master key

#### Return

- `ESP_OK` : succeed
- `ESP_ERR_ESPNOW_NOT_INIT` : ESPNOW is not initialized
- `ESP_ERR_ESPNOW_ARG` : invalid argument

#### Parameters

- `pmk`: primary master key

## Structures

`struct esp_now_peer_info`  
ESPNOW peer information parameters.

### Public Members

`uint8_t peer_addr[ESP_NOW_ETH_ALEN]`  
ESPNOW peer MAC address that is also the MAC address of station or softap

`uint8_t lmk[ESP_NOW_KEY_LEN]`  
ESPNOW peer local master key that is used to encrypt data

`uint8_t channel`  
Wi-Fi channel that peer uses to send/receive ESPNOW data. If the value is 0, use the current channel which station or softap is on. Otherwise, it must be set as the channel that station or softap is on.

`wifi_interface_t ifidx`  
Wi-Fi interface that peer uses to send/receive ESPNOW data

`bool encrypt`  
ESPNOW data that this peer sends/receives is encrypted or not

`void *priv`  
ESPNOW peer private data

**struct esp\_now\_peer\_num**  
Number of ESPNOW peers which exist currently.

### Public Members

int **total\_num**  
Total number of ESPNOW peers, maximum value is ESP\_NOW\_MAX\_TOTAL\_PEER\_NUM

int **encrypt\_num**  
Number of encrypted ESPNOW peers, maximum value is ESP\_NOW\_MAX\_ENCRYPT\_PEER\_NUM

### Macros

**ESP\_ERR\_ESPNOW\_BASE**  
ESPNOW error number base.

**ESP\_ERR\_ESPNOW\_NOT\_INIT**  
ESPNOW is not initialized.

**ESP\_ERR\_ESPNOW\_ARG**  
Invalid argument

**ESP\_ERR\_ESPNOW\_NO\_MEM**  
Out of memory

**ESP\_ERR\_ESPNOW\_FULL**  
ESPNOW peer list is full

**ESP\_ERR\_ESPNOW\_NOT\_FOUND**  
ESPNOW peer is not found

**ESP\_ERR\_ESPNOW\_INTERNAL**  
Internal error

**ESP\_ERR\_ESPNOW\_EXIST**  
ESPNOW peer has existed

**ESP\_ERR\_ESPNOW\_IF**  
Interface error

**ESP\_NOW\_ETH\_ALEN**  
Length of ESPNOW peer MAC address

**ESP\_NOW\_KEY\_LEN**  
Length of ESPNOW peer local master key

**ESP\_NOW\_MAX\_TOTAL\_PEER\_NUM**  
Maximum number of ESPNOW total peers

**ESP\_NOW\_MAX\_ENCRYPT\_PEER\_NUM**  
Maximum number of ESPNOW encrypted peers

**ESP\_NOW\_MAX\_DATA\_LEN**  
Maximum length of ESPNOW data which is sent very time

## Type Definitions

**typedef struct *esp\_now\_peer\_info* esp\_now\_peer\_info\_t**  
ESPNow peer information parameters.

**typedef struct *esp\_now\_peer\_num* esp\_now\_peer\_num\_t**  
Number of ESPNow peers which exist currently.

**typedef void (\*esp\_now\_recv\_cb\_t) (const uint8\_t \*mac\_addr, const uint8\_t \*data, int data\_len)**  
Callback function of receiving ESPNow data.

### Parameters

- *mac\_addr*: peer MAC address
- *data*: received data
- *data\_len*: length of received data

**typedef void (\*esp\_now\_send\_cb\_t) (const uint8\_t \*mac\_addr, *esp\_now\_send\_status\_t* status)**  
Callback function of sending ESPNow data.

### Parameters

- *mac\_addr*: peer MAC address
- *status*: status of sending ESPNow data (succeed or fail)

## Enumerations

**enum esp\_now\_send\_status\_t**  
Status of sending ESPNow data .

*Values:*

**ESP\_NOW\_SEND\_SUCCESS = 0**  
Send ESPNow data successfully

**ESP\_NOW\_SEND\_FAIL**  
Send ESPNow data fail

Example code for this API section is provided in [wifi](#) directory of ESP-IDF examples.

## 2.2 Bluetooth API

### 2.2.1 Controller && VHCI

#### Overview

#### Instructions

## Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following application:

- This is a BLE advertising demo with virtual HCI interface. Send `Reset/ADV_PARAM/ADV_DATA/ADV_ENABLE` HCI command for BLE advertising - `bluetooth/ble_adv`.

## API Reference

### Header File

- `bt/include/bt.h`

## 2.2.2 BT COMMON

### BT GENERIC DEFINES

#### Overview

Instructions

#### Application Example

Instructions

#### API Reference

##### Header File

- `bt/bluedroid/api/include/esp_bt_defs.h`

##### Structures

**struct esp\_bt\_uuid\_t**

UUID type.

##### Public Members

**uint16\_t len**

UUID length, 16bit, 32bit or 128bit

**union esp\_bt\_uuid\_t::[anonymous] uuid**

UUID

## Macros

**ESP\_BLUEDROID\_STATUS\_CHECK** (status)

**ESP\_BT\_OCTET16\_LEN**

**ESP\_BT\_OCTET8\_LEN**

**ESP\_DEFAULT\_GATT\_IF**

Default GATT interface id.

**ESP\_BLE\_CONN\_INT\_MIN**

relate to BTM\_BLE\_CONN\_INT\_MIN in btm\_ble\_api.h

**ESP\_BLE\_CONN\_INT\_MAX**

relate to BTM\_BLE\_CONN\_INT\_MAX in btm\_ble\_api.h

**ESP\_BLE\_CONN\_LATENCY\_MAX**

relate to ESP\_BLE\_CONN\_LATENCY\_MAX in btm\_ble\_api.h

**ESP\_BLE\_CONN\_SUP\_TOUT\_MIN**

relate to BTM\_BLE\_CONN\_SUP\_TOUT\_MIN in btm\_ble\_api.h

**ESP\_BLE\_CONN\_SUP\_TOUT\_MAX**

relate to ESP\_BLE\_CONN\_SUP\_TOUT\_MAX in btm\_ble\_api.h

**ESP\_BLE\_CONN\_PARAM\_UNDEF**

**ESP\_BLE\_SCAN\_PARAM\_UNDEF**

**ESP\_BLE\_IS\_VALID\_PARAM** (x, min, max)

Check the param is valid or not.

**ESP\_UUID\_LEN\_16**

**ESP\_UUID\_LEN\_32**

**ESP\_UUID\_LEN\_128**

**ESP\_BD\_ADDR\_LEN**

Bluetooth address length.

**ESP\_BLE\_ENC\_KEY\_MASK**

Used to exchange the encryption key in the init key & response key.

**ESP\_BLE\_ID\_KEY\_MASK**

Used to exchange the IRK key in the init key & response key.

**ESP\_BLE\_CSR\_KEY\_MASK**

Used to exchange the CSRK key in the init key & response key.

**ESP\_BLE\_LINK\_KEY\_MASK**

Used to exchange the link key(this key just used in the BLE & BR/EDR coexist mode) in the init key & response key.

**ESP\_APP\_ID\_MIN**

Minimum of the application id.

**ESP\_APP\_ID\_MAX**

Maximum of the application id.

**ESP\_BD\_ADDR\_STR**

**ESP\_BD\_ADDR\_HEX** (addr)

## Type Definitions

```
typedef uint8_t esp_bt_octet16_t[ESP_BT_OCTET16_LEN]
typedef uint8_t esp_bt_octet8_t[ESP_BT_OCTET8_LEN]
typedef uint8_t esp_link_key[ESP_BT_OCTET16_LEN]
typedef uint8_t esp_bd_addr_t[ESP_BT_ADDR_LEN]
    Bluetooth device address.
typedef uint8_t esp_ble_key_mask_t
```

## Enumerations

```
enum esp_bt_status_t
```

Status Return Value.

*Values:*

```
ESP_BT_STATUS_SUCCESS = 0
ESP_BT_STATUS_FAIL
ESP_BT_STATUS_NOT_READY
ESP_BT_STATUS_NOMEM
ESP_BT_STATUS_BUSY
ESP_BT_STATUS_DONE = 5
ESP_BT_STATUS_UNSUPPORTED
ESP_BT_STATUS_PARM_INVALID
ESP_BT_STATUS_UNHANDLED
ESP_BT_STATUS_AUTH_FAILURE
ESP_BT_STATUS_RMT_DEV_DOWN = 10
ESP_BT_STATUS_AUTH_REJECTED
ESP_BT_STATUS_INVALID_STATIC_RAND_ADDR
ESP_BT_STATUS_PENDING
ESP_BT_STATUS_UNACCEPT_CONN_INTERVAL
ESP_BT_STATUS_PARAM_OUT_OF_RANGE
ESP_BT_STATUS_TIMEOUT
ESP_BT_STATUS_PEER_LE_DATA_LEN_UNSUPPORTED
ESP_BT_STATUS_CONTROL_LE_DATA_LEN_UNSUPPORTED
ESP_BT_STATUS_ERR_ILLEGAL_PARAMETER_FMT
ESP_BT_STATUS_MEMORY_FULL
```

```
enum esp_bt_dev_type_t
```

Bluetooth device type.

*Values:*

`ESP_BT_DEVICE_TYPE_BREDR = 0x01`

`ESP_BT_DEVICE_TYPE_BLE = 0x02`

`ESP_BT_DEVICE_TYPE_DUMO = 0x03`

`enum esp_ble_addr_type_t`

BLE device address type.

*Values:*

`BLE_ADDR_TYPE_PUBLIC = 0x00`

`BLE_ADDR_TYPE_RANDOM = 0x01`

`BLE_ADDR_TYPE_RPA_PUBLIC = 0x02`

`BLE_ADDR_TYPE_RPA_RANDOM = 0x03`

## BT MAIN API

### Overview

Instructions

### Application Example

Instructions

### API Reference

#### Header File

- `bt/bluedroid/api/include/esp_bt_main.h`

#### Functions

`esp_bluedroid_status_t esp_bluedroid_get_status` (void)

Get bluetooth stack status.

**Return** Bluetooth stack status

`esp_err_t esp_bluedroid_enable` (void)

Enable bluetooth, must after `esp_bluedroid_init()`

**Return**

- `ESP_OK` : Succeed
- Other : Failed

`esp_err_t esp_bluedroid_disable` (void)

Disable bluetooth, must prior to `esp_bluedroid_deinit()`

**Return**

- ESP\_OK : Succeed
- Other : Failed

esp\_err\_t **esp\_bluedroid\_init** (void)

Init and alloc the resource for bluetooth, must be prior to every bluetooth stuff.

**Return**

- ESP\_OK : Succeed
- Other : Failed

esp\_err\_t **esp\_bluedroid\_deinit** (void)

Deinit and free the resource for bluetooth, must be after every bluetooth stuff.

**Return**

- ESP\_OK : Succeed
- Other : Failed

## Enumerations

enum **esp\_bluedroid\_status\_t**

Bluetooth stack status type, to indicate whether the bluetooth stack is ready.

*Values:*

**ESP\_BLUEDROID\_STATUS\_UNINITIALIZED** = 0

Bluetooth not initialized

**ESP\_BLUEDROID\_STATUS\_INITIALIZED**

Bluetooth initialized but not enabled

**ESP\_BLUEDROID\_STATUS\_ENABLED**

Bluetooth initialized and enabled

## BT DEVICE APIS

### Overview

Bluetooth device reference APIs.

[Instructions](#)

### Application Example

[Instructions](#)

### API Reference

#### Header File

- [bt/bluedroid/api/include/esp\\_bt\\_device.h](#)

## Functions

**const** uint8\_t \***esp\_bt\_dev\_get\_address** (void)

Get bluetooth device address. Must use after “esp\_bluedroid\_enable”.

**Return** bluetooth device address (six bytes), or NULL if bluetooth stack is not enabled

esp\_err\_t **esp\_bt\_dev\_set\_device\_name** (const char \*name)

Set bluetooth device name. This function should be called after esp\_bluedroid\_enable() completes successfully.

### Return

- ESP\_OK : Succeed
- ESP\_ERR\_INVALID\_ARG : if name is NULL pointer or empty, or string length out of limit
- ESP\_INVALID\_STATE : if bluetooth stack is not yet enabled
- ESP\_FAIL : others

### Parameters

- name: : device name to be set

## 2.2.3 BT LE

### GAP API

#### Overview

Instructions

#### Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following applications:

- The two demos use different GAP APIs, such like advertising, scan, set device name and others - `bluetooth/gatt_server`, `bluetooth/gatt_client`

### API Reference

#### Header File

- `bt/bluedroid/api/include/esp_gap_ble_api.h`

## Functions

esp\_err\_t **esp\_ble\_gap\_register\_callback** (*esp\_gap\_ble\_cb\_t* callback)

This function is called to occur gap event, such as scan result.

### Return

- ESP\_OK : success

- other : failed

#### Parameters

- callback: callback function

esp\_err\_t **esp\_ble\_gap\_config\_adv\_data** (*esp\_ble\_adv\_data\_t* \*adv\_data)

This function is called to override the BTA default ADV parameters.

#### Return

- ESP\_OK : success
- other : failed

#### Parameters

- adv\_data: Pointer to User defined ADV data structure. This memory space can not be freed until callback of config\_adv\_data is received.

esp\_err\_t **esp\_ble\_gap\_set\_scan\_params** (*esp\_ble\_scan\_params\_t* \*scan\_params)

This function is called to set scan parameters.

#### Return

- ESP\_OK : success
- other : failed

#### Parameters

- scan\_params: Pointer to User defined scan\_params data structure. This memory space can not be freed until callback of set\_scan\_params

esp\_err\_t **esp\_ble\_gap\_start\_scanning** (uint32\_t duration)

This procedure keep the device scanning the peer device which advertising on the air.

#### Return

- ESP\_OK : success
- other : failed

#### Parameters

- duration: Keeping the scanning time, the unit is second.

esp\_err\_t **esp\_ble\_gap\_stop\_scanning** (void)

This function call to stop the device scanning the peer device which advertising on the air.

#### Return

- ESP\_OK : success
- other : failed

esp\_err\_t **esp\_ble\_gap\_start\_advertising** (*esp\_ble\_adv\_params\_t* \*adv\_params)

This function is called to start advertising.

#### Return

- ESP\_OK : success

- other : failed

#### Parameters

- `adv_params`: pointer to User defined `adv_params` data structure.

`esp_err_t esp_ble_gap_stop_advertising` (void)

This function is called to stop advertising.

#### Return

- `ESP_OK` : success
- other : failed

`esp_err_t esp_ble_gap_update_conn_params` (`esp_ble_conn_update_params_t *params`)

Update connection parameters, can only be used when connection is up.

#### Return

- `ESP_OK` : success
- other : failed

#### Parameters

- `params`: - connection update parameters

`esp_err_t esp_ble_gap_set_pkt_data_len` (`esp_bd_addr_t remote_device`, `uint16_t tx_data_length`)

This function is to set maximum LE data packet size.

#### Return

- `ESP_OK` : success
- other : failed

`esp_err_t esp_ble_gap_set_rand_addr` (`esp_bd_addr_t rand_addr`)

This function set the random address for the application.

#### Return

- `ESP_OK` : success
- other : failed

#### Parameters

- `rand_addr`: the random address which should be setting

`esp_err_t esp_ble_gap_config_local_privacy` (bool `privacy_enable`)

Enable/disable privacy on the local device.

#### Return

- `ESP_OK` : success
- other : failed

#### Parameters

- `privacy_enable`: - enable/disable privacy on remote device.

`esp_err_t esp_ble_gap_update_whitelist` (bool *add\_remove*, *esp\_bd\_addr\_t remote\_bda*)  
 Add or remove device from white list.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- *add\_remove*: the value is true if added the ble device to the white list, and false remove to the white list.
- *remote\_bda*: the remote device address add/remove from the white list.

`esp_err_t esp_ble_gap_get_whitelist_size` (uint16\_t \**length*)  
 Get the whitelist size in the controller.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- *length*: the white list length.

`esp_err_t esp_ble_gap_set_prefer_conn_params` (*esp\_bd\_addr\_t bd\_addr*, uint16\_t *min\_conn\_int*,  
 uint16\_t *max\_conn\_int*, uint16\_t *slave\_latency*,  
 uint16\_t *supervision\_tout*)

This function is called to set the preferred connection parameters when default connection parameter is not desired before connecting. This API can only be used in the master role.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- *bd\_addr*: BD address of the peripheral
- *min\_conn\_int*: minimum preferred connection interval
- *max\_conn\_int*: maximum preferred connection interval
- *slave\_latency*: preferred slave latency
- *supervision\_tout*: preferred supervision timeout

`esp_err_t esp_ble_gap_set_device_name` (const char \**name*)  
 Set device name to the local device.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- *name*: - device name.

`uint8_t *esp_ble_resolve_adv_data (uint8_t *adv_data, uint8_t type, uint8_t *length)`

This function is called to get ADV data for a specific type.

**Return** - ESP\_OK : success

- other : failed

**Parameters**

- `adv_data`: - pointer of ADV data which to be resolved
- `type`: - finding ADV data type
- `length`: - return the length of ADV data not including type

`esp_err_t esp_ble_gap_config_adv_data_raw (uint8_t *raw_data, uint32_t raw_data_len)`

This function is called to set raw advertising data. User need to fill ADV data by self.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- `raw_data`: : raw advertising data
- `raw_data_len`: : raw advertising data length , less than 31 bytes

`esp_err_t esp_ble_gap_config_scan_rsp_data_raw (uint8_t *raw_data, uint32_t raw_data_len)`

This function is called to set raw scan response data. User need to fill scan response data by self.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- `raw_data`: : raw scan response data
- `raw_data_len`: : raw scan response data length , less than 31 bytes

`esp_err_t esp_ble_gap_read_rssi (esp_bd_addr_t remote_addr)`

This function is called to read the RSSI of remote device. The address of link policy results are returned in the gap callback function with ESP\_GAP\_BLE\_READ\_RSSI\_COMPLETE\_EVT event.

**Return**

- ESP\_OK : success
- other : failed

**Parameters**

- `remote_addr`: : The remote connection device address.

`esp_err_t esp_ble_gap_set_security_param (esp_ble_sm_param_t param_type, void *value, uint8_t len)`

Set a GAP security parameter value. Overrides the default value.

**Return** - ESP\_OK : success

- other : failed

#### Parameters

- `param_type`: : the type of the param which to be set
- `value`: : the param value
- `len`: : the length of the param value

`esp_err_t esp_ble_gap_security_rsp` (*esp\_bd\_addr\_t* `bd_addr`, *bool* `accept`)  
Grant security request access.

**Return** - ESP\_OK : success

- other : failed

#### Parameters

- `bd_addr`: : BD address of the peer
- `accept`: : accept the security request or not

`esp_err_t esp_ble_set_encryption` (*esp\_bd\_addr\_t* `bd_addr`, *esp\_ble\_sec\_act\_t* `sec_act`)  
Set a gap parameter value. Use this function to change the default GAP parameter values.

**Return** - ESP\_OK : success

- other : failed

#### Parameters

- `bd_addr`: : the address of the peer device need to encryption
- `sec_act`: : This is the security action to indicate what kind of BLE security level is required for the BLE link if the BLE is supported

`esp_err_t esp_ble_passkey_reply` (*esp\_bd\_addr\_t* `bd_addr`, *bool* `accept`, *uint32\_t* `passkey`)  
Reply the key value to the peer device in the lasecy connection stage.

**Return** - ESP\_OK : success

- other : failed

#### Parameters

- `bd_addr`: : BD address of the peer
- `accept`: : passkey entry sucessful or declined.
- `passkey`: : passkey value, must be a 6 digit number, can be lead by 0.

`esp_err_t esp_ble_confirm_reply` (*esp\_bd\_addr\_t* `bd_addr`, *bool* `accept`)  
Reply the comfirm value to the peer device in the lasecy connection stage.

**Return** - ESP\_OK : success

- other : failed

#### Parameters

- `bd_addr`: : BD address of the peer device
- `accept`: : numbers to compare are the same or different.

`esp_err_t esp_ble_remove_bond_device(esp_bd_addr_t bd_addr)`

Removes a device from the security database list of peer device. It manages unpairing event while connected.

**Return** - ESP\_OK : success

- other : failed

**Parameters**

- `bd_addr`: : BD address of the peer device

`int esp_ble_get_bond_device_num(void)`

Get the device number from the security database list of peer device. It will return the device bonded number immediately.

**Return** -  $\geq 0$  : bonded devices number.

- $< 0$  : failed

`esp_err_t esp_ble_get_bond_device_list(int *dev_num, esp_ble_bond_dev_t *dev_list)`

Get the device from the security database list of peer device. It will return the device bonded information immediately.

**Return** - ESP\_OK : success

- other : failed

**Parameters**

- `dev_num`: Indicate the `dev_list` array(buffer) size as input. If `dev_num` is large enough, it means the actual number as output. Suggest that `dev_num` value equal to `esp_ble_get_bond_device_num()`.
- `dev_list`: an array(buffer) of `esp_ble_bond_dev_t` type. Use for storing the bonded devices address. The `dev_list` should be allocated by who call this API.

`esp_err_t esp_ble_gap_disconnect(esp_bd_addr_t remote_device)`

This function is to disconnect the physical connection of the peer device gattc maybe have multiple virtual GATT server connections when multiple `app_id` registered. `esp_ble_gattc_close(esp_gatt_if_t gattc_if, uint16_t conn_id)` only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. `esp_ble_gap_disconnect(esp_bd_addr_t remote_device)` disconnect the physical connection directly.

**Return** - ESP\_OK : success

- other : failed

**Parameters**

- `remote_device`: : BD address of the peer device

## Unions

`union esp_ble_key_value_t`

`#include <esp_gap_ble_api.h>` union type of the security key value

## Public Members

*esp\_ble\_penc\_keys\_t* **penc\_key**  
received peer encryption key

*esp\_ble\_pcsrkeys\_t* **pcsrkey**  
received peer device SRK

*esp\_ble\_pidkeys\_t* **pidkey**  
peer device ID key

*esp\_ble\_lenckeys\_t* **lenckey**  
local encryption reproduction keys LTK == d1(ER,DIV,0)

*esp\_ble\_lcsrkeys\_t* **lcsrkey**  
local device CSRK = d1(ER,DIV,1)

**union esp\_ble\_sec\_t**  
*#include <esp\_gap\_ble\_api.h>* union associated with ble security

## Public Members

*esp\_ble\_sec\_key\_notif\_t* **key\_notif**  
passkey notification

*esp\_ble\_sec\_req\_t* **ble\_req**  
BLE SMP related request

*esp\_ble\_key\_t* **ble\_key**  
BLE SMP keys used when pairing

*esp\_ble\_local\_id\_keys\_t* **ble\_id\_keys**  
BLE IR event

*esp\_ble\_auth\_cmpl\_t* **auth\_cmpl**  
Authentication complete indication.

**union esp\_ble\_gap\_cb\_param\_t**  
*#include <esp\_gap\_ble\_api.h>* Gap callback parameters union.

## Public Members

**struct esp\_ble\_gap\_cb\_param\_t::ble\_adv\_data\_cmpl\_evt\_param** **adv\_data\_cmpl**  
Event parameter of ESP\_GAP\_BLE\_ADV\_DATA\_SET\_COMPLETE\_EVT

**struct esp\_ble\_gap\_cb\_param\_t::ble\_scan\_rsp\_data\_cmpl\_evt\_param** **scan\_rsp\_data\_cmpl**  
Event parameter of ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_SET\_COMPLETE\_EVT

**struct esp\_ble\_gap\_cb\_param\_t::ble\_scan\_param\_cmpl\_evt\_param** **scan\_param\_cmpl**  
Event parameter of ESP\_GAP\_BLE\_SCAN\_PARAM\_SET\_COMPLETE\_EVT

**struct esp\_ble\_gap\_cb\_param\_t::ble\_scan\_result\_evt\_param** **scan\_rst**  
Event parameter of ESP\_GAP\_BLE\_SCAN\_RESULT\_EVT

**struct esp\_ble\_gap\_cb\_param\_t::ble\_adv\_data\_raw\_cmpl\_evt\_param** **adv\_data\_raw\_cmpl**  
Event parameter of ESP\_GAP\_BLE\_ADV\_DATA\_RAW\_SET\_COMPLETE\_EVT

**struct esp\_ble\_gap\_cb\_param\_t::ble\_scan\_rsp\_data\_raw\_cmpl\_evt\_param** **scan\_rsp\_data\_raw\_cmpl**  
Event parameter of ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_RAW\_SET\_COMPLETE\_EVT

```

struct esp_ble_gap_cb_param_t::ble_adv_start_cmpl_evt_param adv_start_cmpl
    Event parameter of ESP_GAP_BLE_ADV_START_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_param scan_start_cmpl
    Event parameter of ESP_GAP_BLE_SCAN_START_COMPLETE_EVT

esp_ble_sec_t ble_security
    ble gap security union type

struct esp_ble_gap_cb_param_t::ble_scan_stop_cmpl_evt_param scan_stop_cmpl
    Event parameter of ESP_GAP_BLE_SCAN_STOP_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_adv_stop_cmpl_evt_param adv_stop_cmpl
    Event parameter of ESP_GAP_BLE_ADV_STOP_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_set_rand_cmpl_evt_param set_rand_addr_cmpl
    Event parameter of ESP_GAP_BLE_SET_STATIC_RAND_ADDR_EVT

struct esp_ble_gap_cb_param_t::ble_update_conn_params_evt_param update_conn_params
    Event parameter of ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT

struct esp_ble_gap_cb_param_t::ble_pkt_data_length_cmpl_evt_param pkt_data_lenth_cmpl
    Event parameter of ESP_GAP_BLE_SET_PKT_LENGTH_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_local_privacy_cmpl_evt_param local_privacy_cmpl
    Event parameter of ESP_GAP_BLE_SET_LOCAL_PRIVACY_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_remove_bond_dev_cmpl_evt_param remove_bond_dev_cmpl
    Event parameter of ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_clear_bond_dev_cmpl_evt_param clear_bond_dev_cmpl
    Event parameter of ESP_GAP_BLE_CLEAR_BOND_DEV_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_get_bond_dev_cmpl_evt_param get_bond_dev_cmpl
    Event parameter of ESP_GAP_BLE_GET_BOND_DEV_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_read_rssi_cmpl_evt_param read_rssi_cmpl
    Event parameter of ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT

struct esp_ble_gap_cb_param_t::ble_update_whitelist_cmpl_evt_param update_whitelist_cmpl
    Event parameter of ESP_GAP_BLE_UPDATE_WHITELIST_COMPLETE_EVT

struct ble_adv_data_cmpl_evt_param
    #include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT.

```

## Public Members

```

esp_bt_status_t status
    Indicate the set advertising data operation success status

```

```

struct ble_adv_data_raw_cmpl_evt_param
    #include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT.

```

## Public Members

```

esp_bt_status_t status
    Indicate the set raw advertising data operation success status

```

```

struct ble_adv_start_cmpl_evt_param
    #include <esp_gap_ble_api.h> ESP_GAP_BLE_ADV_START_COMPLETE_EVT.

```

### Public Members

*esp\_bt\_status\_t* **status**

Indicate advertising start operation success status

**struct ble\_adv\_stop\_cmpl\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_ADV\_STOP\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**

Indicate adv stop operation success status

**struct ble\_clear\_bond\_dev\_cmpl\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_CLEAR\_BOND\_DEV\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the clear bond device operation success status

**struct ble\_get\_bond\_dev\_cmpl\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_GET\_BOND\_DEV\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the get bond device operation success status

*uint8\_t* **dev\_num**

Indicate the get number device in the bond list

*esp\_ble\_bond\_dev\_t* \***bond\_dev**

the pointer to the bond device Structure

**struct ble\_local\_privacy\_cmpl\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SET\_LOCAL\_PRIVACY\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the set local privacy operation success status

**struct ble\_pkt\_data\_length\_cmpl\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SET\_PKT\_LENGTH\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the set pkt data length operation success status

*esp\_ble\_pkt\_data\_length\_params\_t* **params**

pkt data length value

```
struct ble_read_rssi_cmpl_evt_param  
#include <esp_gap_ble_api.h> ESP_GAP_BLE_READ_RSSI_COMPLETE_EVT.
```

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the read adv tx power operation success status

**int8\_t rssi**

The ble remote device rssi value, the range is from -127 to 20, the unit is dbm, if the RSSI cannot be read, the RSSI metric shall be set to 127.

*esp\_bd\_addr\_t* **remote\_addr**

The remote device address

```
struct ble_remove_bond_dev_cmpl_evt_param  
#include <esp_gap_ble_api.h> ESP_GAP_BLE_REMOVE_BOND_DEV_COMPLETE_EVT.
```

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the remove bond device operation success status

*esp\_bd\_addr\_t* **bd\_addr**

The device address which has been remove from the bond list

```
struct ble_scan_param_cmpl_evt_param  
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT.
```

### Public Members

*esp\_bt\_status\_t* **status**

Indicate the set scan param operation success status

```
struct ble_scan_result_evt_param  
#include <esp_gap_ble_api.h> ESP_GAP_BLE_SCAN_RESULT_EVT.
```

### Public Members

*esp\_gap\_search\_evt\_t* **search\_evt**

Search event type

*esp\_bd\_addr\_t* **bda**

Bluetooth device address which has been searched

*esp\_bt\_dev\_type\_t* **dev\_type**

Device type

*esp\_ble\_addr\_type\_t* **ble\_addr\_type**

Ble device address type

*esp\_ble\_evt\_type\_t* **ble\_evt\_type**

Ble scan result event type

**int rssi**

Searched device's RSSI

uint8\_t **ble\_adv**[ESP\_BLE\_ADV\_DATA\_LEN\_MAX + ESP\_BLE\_SCAN\_RSP\_DATA\_LEN\_MAX]  
Received EIR

int **flag**  
Advertising data flag bit

int **num\_resps**  
Scan result number

uint8\_t **adv\_data\_len**  
Adv data length

uint8\_t **scan\_rsp\_len**  
Scan response length

**struct ble\_scan\_rsp\_data\_cmpl\_evt\_param**  
*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_SET\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**  
Indicate the set scan response data operation success status

**struct ble\_scan\_rsp\_data\_raw\_cmpl\_evt\_param**  
*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_RAW\_SET\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**  
Indicate the set raw advertising data operation success status

**struct ble\_scan\_start\_cmpl\_evt\_param**  
*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SCAN\_START\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**  
Indicate scan start operation success status

**struct ble\_scan\_stop\_cmpl\_evt\_param**  
*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SCAN\_STOP\_COMPLETE\_EVT.

### Public Members

*esp\_bt\_status\_t* **status**  
Indicate scan stop operation success status

**struct ble\_set\_rand\_cmpl\_evt\_param**  
*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_SET\_STATIC\_RAND\_ADDR\_EVT.

## Public Members

*esp\_bt\_status\_t* **status**

Indicate set static rand address operation success status

**struct ble\_update\_conn\_params\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_UPDATE\_CONN\_PARAMS\_EVT.

## Public Members

*esp\_bt\_status\_t* **status**

Indicate update connection parameters success status

*esp\_bd\_addr\_t* **bda**

Bluetooth device address

uint16\_t **min\_int**

Min connection interval

uint16\_t **max\_int**

Max connection interval

uint16\_t **latency**

Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

uint16\_t **conn\_int**

Current connection interval

uint16\_t **timeout**

Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N \* 10 msec

**struct ble\_update\_whitelist\_cmpl\_evt\_param**

*#include <esp\_gap\_ble\_api.h>* ESP\_GAP\_BLE\_UPDATE\_WHITELIST\_COMPLETE\_EVT.

## Public Members

*esp\_bt\_status\_t* **status**

Indicate the add or remove whitelist operation success status

*esp\_ble\_wl\_opration\_t* **wl\_opration**

The value is ESP\_BLE\_WHITELIST\_ADD if add address to whitelist operation success, ESP\_BLE\_WHITELIST\_REMOVE if remove address from the whitelist operation success

## Structures

**struct esp\_ble\_adv\_params\_t**

Advertising parameters.

## Public Members

uint16\_t **adv\_int\_min**

Minimum advertising interval for undirected and low duty cycle directed advertising. Range: 0x0020 to 0x4000 Default: N = 0x0800 (1.28 second) Time = N \* 0.625 msec Time Range: 20 ms to 10.24 sec

`uint16_t adv_int_max`  
 Maximum advertising interval for undirected and low duty cycle directed advertising. Range: 0x0020 to 0x4000 Default: N = 0x0800 (1.28 second) Time = N \* 0.625 msec Time Range: 20 ms to 10.24 sec Advertising max interval

`esp_ble_adv_type_t adv_type`  
 Advertising type

`esp_ble_addr_type_t own_addr_type`  
 Owner bluetooth device address type

`esp_bd_addr_t peer_addr`  
 Peer device bluetooth device address

`esp_ble_addr_type_t peer_addr_type`  
 Peer device bluetooth device address type, only support public address type and random address type

`esp_ble_adv_channel_t channel_map`  
 Advertising channel map

`esp_ble_adv_filter_t adv_filter_policy`  
 Advertising filter policy

`struct esp_ble_adv_data_t`

Advertising data content, according to “Supplement to the Bluetooth Core Specification”.

### Public Members

bool `set_scan_rsp`  
 Set this advertising data as scan response or not

bool `include_name`  
 Advertising data include device name or not

bool `include_txpower`  
 Advertising data include TX power

int `min_interval`  
 Advertising data show advertising min interval

int `max_interval`  
 Advertising data show advertising max interval

int `appearance`  
 External appearance of device

uint16\_t `manufacturer_len`  
 Manufacturer data length

uint8\_t \*`p_manufacturer_data`  
 Manufacturer data point

uint16\_t `service_data_len`  
 Service data length

uint8\_t \*`p_service_data`  
 Service data point

uint16\_t `service_uuid_len`  
 Service uuid length

`uint8_t *p_service_uuid`  
Service uuid array point

`uint8_t flag`  
Advertising flag of discovery mode, see BLE\_ADV\_DATA\_FLAG detail

**struct esp\_ble\_scan\_params\_t**  
Ble scan parameters.

### Public Members

`esp_ble_scan_type_t scan_type`  
Scan type

`esp_ble_addr_type_t own_addr_type`  
Owner address type

`esp_ble_scan_filter_t scan_filter_policy`  
Scan filter policy

`uint16_t scan_interval`  
Scan interval. This is defined as the time interval from when the Controller started its last LE scan until it begins the subsequent LE scan. Range: 0x0004 to 0x4000 Default: 0x0010 (10 ms) Time = N \* 0.625 msec Time Range: 2.5 msec to 10.24 seconds

`uint16_t scan_window`  
Scan window. The duration of the LE scan. LE\_Scan\_Window shall be less than or equal to LE\_Scan\_Interval Range: 0x0004 to 0x4000 Default: 0x0010 (10 ms) Time = N \* 0.625 msec Time Range: 2.5 msec to 10240 msec

`esp_ble_scan_duplicate_t scan_duplicate`  
The Scan\_Duplicates parameter controls whether the Link Layer should filter out duplicate advertising reports (BLE\_SCAN\_DUPLICATE\_ENABLE) to the Host, or if the Link Layer should generate advertising reports for each packet received

**struct esp\_ble\_conn\_update\_params\_t**  
Connection update parameters.

### Public Members

`esp_bd_addr_t bda`  
Bluetooth device address

`uint16_t min_int`  
Min connection interval

`uint16_t max_int`  
Max connection interval

`uint16_t latency`  
Slave latency for the connection in number of connection events. Range: 0x0000 to 0x01F3

`uint16_t timeout`  
Supervision timeout for the LE Link. Range: 0x000A to 0x0C80. Mandatory Range: 0x000A to 0x0C80 Time = N \* 10 msec Time Range: 100 msec to 32 seconds

**struct esp\_ble\_pkt\_data\_length\_params\_t**  
BLE pkt data length keys.

### Public Members

`uint16_t rx_len`  
pkt rx data length value

`uint16_t tx_len`  
pkt tx data length value

**struct esp\_ble\_penc\_keys\_t**  
BLE encryption keys.

### Public Members

`esp_bt_octet16_t ltk`  
The long term key

`esp_bt_octet8_t rand`  
The random number

`uint16_t ediv`  
The ediv value

`uint8_t sec_level`  
The security level of the security link

`uint8_t key_size`  
The key size(7~16) of the security link

**struct esp\_ble\_pcsrkeys\_t**  
BLE CSRK keys.

### Public Members

`uint32_t counter`  
The counter

`esp_bt_octet16_t csrkey`  
The csrkey

`uint8_t sec_level`  
The security level

**struct esp\_ble\_pidkeys\_t**  
BLE pid keys.

### Public Members

`esp_bt_octet16_t irk`  
The irk value

`esp_ble_addr_type_t addr_type`  
The address type

`esp_bd_addr_t static_addr`  
The static address

**struct esp\_ble\_lenc\_keys\_t**  
BLE Encryption reproduction keys.

### Public Members

*esp\_bt\_octet16\_t* **ltk**  
The long term key

**uint16\_t div**  
The div value

**uint8\_t key\_size**  
The key size of the security link

**uint8\_t sec\_level**  
The security level of the security link

**struct esp\_ble\_lcsrkeys**  
BLE SRK keys.

### Public Members

**uint32\_t counter**  
The counter value

**uint16\_t div**  
The div value

**uint8\_t sec\_level**  
The security level of the security link

*esp\_bt\_octet16\_t* **csrkey**  
The csrkey value

**struct esp\_ble\_sec\_key\_notif\_t**  
Structure associated with ESP\_KEY\_NOTIF\_EVT.

### Public Members

*esp\_bd\_addr\_t* **bd\_addr**  
peer address

**uint32\_t passkey**  
the numeric value for comparison. If `just_works`, do not show this number to UI

**struct esp\_ble\_sec\_req\_t**  
Structure of the security request.

### Public Members

*esp\_bd\_addr\_t* **bd\_addr**  
peer address

**struct esp\_ble\_bond\_key\_info\_t**  
struct type of the bond key informatuon value

### Public Members

*esp\_ble\_key\_mask\_t* **key\_mask**  
the key mask to indicate witch key is present

*esp\_ble\_penc\_keys\_t* **penc\_key**  
received peer encryption key

*esp\_ble\_pcsrkeys\_t* **pcsrk\_key**  
received peer device SRK

*esp\_ble\_pid\_keys\_t* **pid\_key**  
peer device ID key

**struct esp\_ble\_bond\_dev\_t**  
struct type of the bond device value

### Public Members

*esp\_bd\_addr\_t* **bd\_addr**  
peer address

*esp\_ble\_bond\_key\_info\_t* **bond\_key**  
the bond key information

**struct esp\_ble\_key\_t**  
union type of the security key value

### Public Members

*esp\_bd\_addr\_t* **bd\_addr**  
peer address

*esp\_ble\_key\_type\_t* **key\_type**  
key type of the security link

*esp\_ble\_key\_value\_t* **p\_key\_value**  
the pointer to the key value

**struct esp\_ble\_local\_id\_keys\_t**  
structure type of the ble local id keys value

### Public Members

*esp\_bt\_octet16\_t* **ir**  
the 16 bits of the ir value

*esp\_bt\_octet16\_t* **irk**  
the 16 bits of the ir key value

*esp\_bt\_octet16\_t* **dhk**  
the 16 bits of the dh key value

**struct esp\_ble\_auth\_cmpl\_t**  
Structure associated with ESP\_AUTH\_CMPL\_EVT.

## Public Members

- esp\_bd\_addr\_t* **bd\_addr**  
BD address peer device.
- bool **key\_present**  
Valid link key value in key element
- esp\_link\_key* **key**  
Link key associated with peer device.
- uint8\_t **key\_type**  
The type of Link Key
- bool **success**  
TRUE of authentication succeeded, FALSE if failed.
- uint8\_t **fail\_reason**  
The HCI reason/error code for when success=FALSE
- esp\_ble\_addr\_type\_t* **addr\_type**  
Peer device address type
- esp\_bt\_dev\_type\_t* **dev\_type**  
Device type

## Macros

- ESP\_BLE\_ADV\_FLAG\_LIMIT\_DISC**  
BLE\_ADV\_DATA\_FLAG data flag bit definition used for advertising data flag
- ESP\_BLE\_ADV\_FLAG\_GEN\_DISC**
- ESP\_BLE\_ADV\_FLAG\_BREDR\_NOT\_SPT**
- ESP\_BLE\_ADV\_FLAG\_DMT\_CONTROLLER\_SPT**
- ESP\_BLE\_ADV\_FLAG\_DMT\_HOST\_SPT**
- ESP\_BLE\_ADV\_FLAG\_NON\_LIMIT\_DISC**
- ESP\_LE\_KEY\_NONE**
- ESP\_LE\_KEY\_PENC**
- ESP\_LE\_KEY\_PID**
- ESP\_LE\_KEY\_PCSRK**
- ESP\_LE\_KEY\_PLK**
- ESP\_LE\_KEY\_LLK**
- ESP\_LE\_KEY\_LENC**
- ESP\_LE\_KEY\_LID**
- ESP\_LE\_KEY\_LCSRK**
- ESP\_LE\_AUTH\_NO\_BOND**
- ESP\_LE\_AUTH\_BOND**
- ESP\_LE\_AUTH\_REQ\_MITM**

**ESP\_LE\_AUTH\_REQ\_SC\_ONLY**

**ESP\_LE\_AUTH\_REQ\_SC\_BOND**

**ESP\_LE\_AUTH\_REQ\_SC\_MITM**

**ESP\_LE\_AUTH\_REQ\_SC\_MITM\_BOND**

**ESP\_IO\_CAP\_OUT**

**ESP\_IO\_CAP\_IO**

**ESP\_IO\_CAP\_IN**

**ESP\_IO\_CAP\_NONE**

**ESP\_IO\_CAP\_KBDISP**

**ESP\_GAP\_BLE\_ADD\_WHITELIST\_COMPLETE\_EVT**

This is the old name, just for backwards compatibility.

**ESP\_BLE\_ADV\_DATA\_LEN\_MAX**

Advertising data maximum length.

**ESP\_BLE\_SCAN\_RSP\_DATA\_LEN\_MAX**

Scan response data maximum length.

## Type Definitions

```
typedef uint8_t esp_ble_key_type_t
```

```
typedef uint8_t esp_ble_auth_req_t
    combination of the above bit pattern
```

```
typedef uint8_t esp_ble_io_cap_t
    combination of the io capability
```

```
typedef void (*esp_gap_ble_cb_t) (esp_gap_ble_cb_event_t event, esp_ble_gap_cb_param_t
    *param)
```

GAP callback function type.

### Parameters

- event: : Event type
- param: : Point to callback parameter, currently is union type

## Enumerations

```
enum esp_gap_ble_cb_event_t
```

GAP BLE callback event type.

*Values:*

**ESP\_GAP\_BLE\_ADV\_DATA\_SET\_COMPLETE\_EVT = 0**

When advertising data set complete, the event comes

**ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_SET\_COMPLETE\_EVT**

When scan response data set complete, the event comes

**ESP\_GAP\_BLE\_SCAN\_PARAM\_SET\_COMPLETE\_EVT**

When scan parameters set complete, the event comes

**ESP\_GAP\_BLE\_SCAN\_RESULT\_EVT**

When one scan result ready, the event comes each time

**ESP\_GAP\_BLE\_ADV\_DATA\_RAW\_SET\_COMPLETE\_EVT**

When raw advertising data set complete, the event comes

**ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_RAW\_SET\_COMPLETE\_EVT**

When raw advertising data set complete, the event comes

**ESP\_GAP\_BLE\_ADV\_START\_COMPLETE\_EVT**

When start advertising complete, the event comes

**ESP\_GAP\_BLE\_SCAN\_START\_COMPLETE\_EVT**

When start scan complete, the event comes

**ESP\_GAP\_BLE\_AUTH\_CMPL\_EVT**

**ESP\_GAP\_BLE\_KEY\_EVT**

**ESP\_GAP\_BLE\_SEC\_REQ\_EVT**

**ESP\_GAP\_BLE\_PASSKEY\_NOTIF\_EVT**

**ESP\_GAP\_BLE\_PASSKEY\_REQ\_EVT**

**ESP\_GAP\_BLE\_OOB\_REQ\_EVT**

**ESP\_GAP\_BLE\_LOCAL\_IR\_EVT**

**ESP\_GAP\_BLE\_LOCAL\_ER\_EVT**

**ESP\_GAP\_BLE\_NC\_REQ\_EVT**

**ESP\_GAP\_BLE\_ADV\_STOP\_COMPLETE\_EVT**

When stop adv complete, the event comes

**ESP\_GAP\_BLE\_SCAN\_STOP\_COMPLETE\_EVT**

When stop scan complete, the event comes

**ESP\_GAP\_BLE\_SET\_STATIC\_RAND\_ADDR\_EVT**

When set the static rand address complete, the event comes

**ESP\_GAP\_BLE\_UPDATE\_CONN\_PARAMS\_EVT**

When update connection parameters complete, the event comes

**ESP\_GAP\_BLE\_SET\_PKT\_LENGTH\_COMPLETE\_EVT**

When set pkt length complete, the event comes

**ESP\_GAP\_BLE\_SET\_LOCAL\_PRIVACY\_COMPLETE\_EVT**

When Enable/disable privacy on the local device complete, the event comes

**ESP\_GAP\_BLE\_REMOVE\_BOND\_DEV\_COMPLETE\_EVT**

When remove the bond device complete, the event comes

**ESP\_GAP\_BLE\_CLEAR\_BOND\_DEV\_COMPLETE\_EVT**

When clear the bond device clear complete, the event comes

**ESP\_GAP\_BLE\_GET\_BOND\_DEV\_COMPLETE\_EVT**

When get the bond device list complete, the event comes

**ESP\_GAP\_BLE\_READ\_RSSI\_COMPLETE\_EVT**

When read the rssi complete, the event comes

**ESP\_GAP\_BLE\_UPDATE\_WHITELIST\_COMPLETE\_EVT**

When add or remove whitelist complete, the event comes

**ESP\_GAP\_BLE\_EVT\_MAX****enum esp\_ble\_adv\_data\_type**

The type of advertising data(not adv\_type)

*Values:*

```
ESP_BLE_AD_TYPE_FLAG = 0x01
ESP_BLE_AD_TYPE_16SRV_PART = 0x02
ESP_BLE_AD_TYPE_16SRV_CMPL = 0x03
ESP_BLE_AD_TYPE_32SRV_PART = 0x04
ESP_BLE_AD_TYPE_32SRV_CMPL = 0x05
ESP_BLE_AD_TYPE_128SRV_PART = 0x06
ESP_BLE_AD_TYPE_128SRV_CMPL = 0x07
ESP_BLE_AD_TYPE_NAME_SHORT = 0x08
ESP_BLE_AD_TYPE_NAME_CMPL = 0x09
ESP_BLE_AD_TYPE_TX_PWR = 0x0A
ESP_BLE_AD_TYPE_DEV_CLASS = 0x0D
ESP_BLE_AD_TYPE_SM_TK = 0x10
ESP_BLE_AD_TYPE_SM_OOB_FLAG = 0x11
ESP_BLE_AD_TYPE_INT_RANGE = 0x12
ESP_BLE_AD_TYPE_SOL_SRV_UUID = 0x14
ESP_BLE_AD_TYPE_128SOL_SRV_UUID = 0x15
ESP_BLE_AD_TYPE_SERVICE_DATA = 0x16
ESP_BLE_AD_TYPE_PUBLIC_TARGET = 0x17
ESP_BLE_AD_TYPE_RANDOM_TARGET = 0x18
ESP_BLE_AD_TYPE_APPEARANCE = 0x19
ESP_BLE_AD_TYPE_ADV_INT = 0x1A
ESP_BLE_AD_TYPE_LE_DEV_ADDR = 0x1b
ESP_BLE_AD_TYPE_LE_ROLE = 0x1c
ESP_BLE_AD_TYPE_SPAIR_C256 = 0x1d
ESP_BLE_AD_TYPE_SPAIR_R256 = 0x1e
ESP_BLE_AD_TYPE_32SOL_SRV_UUID = 0x1f
ESP_BLE_AD_TYPE_32SERVICE_DATA = 0x20
ESP_BLE_AD_TYPE_128SERVICE_DATA = 0x21
ESP_BLE_AD_TYPE_LE_SECURE_CONFIRM = 0x22
ESP_BLE_AD_TYPE_LE_SECURE_RANDOM = 0x23
ESP_BLE_AD_TYPE_URI = 0x24
ESP_BLE_AD_TYPE_INDOOR_POSITION = 0x25
```

```
ESP_BLE_AD_TYPE_TRANS_DISC_DATA = 0x26
ESP_BLE_AD_TYPE_LE_SUPPORT_FEATURE = 0x27
ESP_BLE_AD_TYPE_CHAN_MAP_UPDATE = 0x28
ESP_BLE_AD_MANUFACTURER_SPECIFIC_TYPE = 0xFF
```

**enum esp\_ble\_adv\_type\_t**  
Advertising mode.

*Values:*

```
ADV_TYPE_IND = 0x00
ADV_TYPE_DIRECT_IND_HIGH = 0x01
ADV_TYPE_SCAN_IND = 0x02
ADV_TYPE_NONCONN_IND = 0x03
ADV_TYPE_DIRECT_IND_LOW = 0x04
```

**enum esp\_ble\_adv\_channel\_t**  
Advertising channel mask.

*Values:*

```
ADV_CHNL_37 = 0x01
ADV_CHNL_38 = 0x02
ADV_CHNL_39 = 0x04
ADV_CHNL_ALL = 0x07
```

**enum esp\_ble\_adv\_filter\_t**

*Values:*

```
ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY = 0x00
    Allow both scan and connection requests from anyone.
ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY
    Allow both scan req from White List devices only and connection req from anyone.
ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST
    Allow both scan req from anyone and connection req from White List devices only.
ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST
    Allow scan and connection requests from White List devices only.
```

**enum esp\_ble\_sec\_act\_t**

*Values:*

```
ESP_BLE_SEC_ENCRYPT = 1
ESP_BLE_SEC_ENCRYPT_NO_MITM
ESP_BLE_SEC_ENCRYPT_MITM
```

**enum esp\_ble\_sm\_param\_t**

*Values:*

```
ESP_BLE_SM_PASSKEY = 0
ESP_BLE_SM_AUTHEN_REQ_MODE
ESP_BLE_SM_IOCAP_MODE
```

**ESP\_BLE\_SM\_SET\_INIT\_KEY****ESP\_BLE\_SM\_SET\_RSP\_KEY****ESP\_BLE\_SM\_MAX\_KEY\_SIZE****enum esp\_ble\_scan\_type\_t**

Ble scan type.

*Values:***BLE\_SCAN\_TYPE\_PASSIVE = 0x0**

Passive scan

**BLE\_SCAN\_TYPE\_ACTIVE = 0x1**

Active scan

**enum esp\_ble\_scan\_filter\_t**

Ble scan filter type.

*Values:***BLE\_SCAN\_FILTER\_ALLOW\_ALL = 0x0**

Accept all :

1. advertisement packets except directed advertising packets not addressed to this device (default).

**BLE\_SCAN\_FILTER\_ALLOW\_ONLY\_WLST = 0x1**

Accept only :

1. advertisement packets from devices where the advertiser's address is in the White list.
2. Directed advertising packets which are not addressed for this device shall be ignored.

**BLE\_SCAN\_FILTER\_ALLOW\_UND\_RPA\_DIR = 0x2**

Accept all :

1. undirected advertisement packets, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

**BLE\_SCAN\_FILTER\_ALLOW\_WLIST\_PRA\_DIR = 0x3**

Accept all :

1. advertisement packets from devices where the advertiser's address is in the White list, and
2. directed advertising packets where the initiator address is a resolvable private address, and
3. directed advertising packets addressed to this device.

**enum esp\_ble\_scan\_duplicate\_t**

Ble scan duplicate type.

*Values:***BLE\_SCAN\_DUPLICATE\_DISABLE = 0x0**

the Link Layer should generate advertising reports to the host for each packet received

**BLE\_SCAN\_DUPLICATE\_ENABLE = 0x1**

the Link Layer should filter out duplicate advertising reports to the Host

**BLE\_SCAN\_DUPLICATE\_MAX = 0x2**

0x02 – 0xFF, Reserved for future use

**enum esp\_gap\_search\_evt\_t**

Sub Event of ESP\_GAP\_BLE\_SCAN\_RESULT\_EVT.

*Values:*

**ESP\_GAP\_SEARCH\_INQ\_RES\_EVT = 0**

Inquiry result for a peer device.

**ESP\_GAP\_SEARCH\_INQ\_CMPL\_EVT = 1**

Inquiry complete.

**ESP\_GAP\_SEARCH\_DISC\_RES\_EVT = 2**

Discovery result for a peer device.

**ESP\_GAP\_SEARCH\_DISC\_BLE\_RES\_EVT = 3**

Discovery result for BLE GATT based service on a peer device.

**ESP\_GAP\_SEARCH\_DISC\_CMPL\_EVT = 4**

Discovery complete.

**ESP\_GAP\_SEARCH\_DI\_DISC\_CMPL\_EVT = 5**

Discovery complete.

**ESP\_GAP\_SEARCH\_SEARCH\_CANCEL\_CMPL\_EVT = 6**

Search cancelled

**enum esp\_ble\_evt\_type\_t**

Ble scan result event type, to indicate the result is scan response or advertising data or other.

*Values:*

**ESP\_BLE\_EVT\_CONN\_ADV = 0x00**

Connectable undirected advertising (ADV\_IND)

**ESP\_BLE\_EVT\_CONN\_DIR\_ADV = 0x01**

Connectable directed advertising (ADV\_DIRECT\_IND)

**ESP\_BLE\_EVT\_DISC\_ADV = 0x02**

Scannable undirected advertising (ADV\_SCAN\_IND)

**ESP\_BLE\_EVT\_NON\_CONN\_ADV = 0x03**

Non connectable undirected advertising (ADV\_NONCONN\_IND)

**ESP\_BLE\_EVT\_SCAN\_RSP = 0x04**

Scan Response (SCAN\_RSP)

**enum esp\_ble\_wl\_opration\_t**

*Values:*

**ESP\_BLE\_WHITELIST\_REMOVE = 0X00**

remove mac from whitelist

**ESP\_BLE\_WHITELIST\_ADD = 0X01**

add address to whitelist

## GATT DEFINES

### Overview

### Instructions

## Application Example

Instructions

## API Reference

### Header File

- `bt/bluedroid/api/include/esp_gatt_defs.h`

### Unions

**union esp\_gatt\_rsp\_t**  
*#include <esp\_gatt\_defs.h>* GATT remote read request response type.

#### Public Members

*esp\_gatt\_value\_t* **attr\_value**  
Gatt attribute structure

`uint16_t` **handle**  
Gatt attribute handle

### Structures

**struct esp\_gatt\_id\_t**  
Gatt id, include uuid and instance id.

#### Public Members

*esp\_bt\_uuid\_t* **uuid**  
UUID

`uint8_t` **inst\_id**  
Instance id

**struct esp\_gatt\_srvc\_id\_t**  
Gatt service id, include id (uuid and instance id) and primary flag.

#### Public Members

*esp\_gatt\_id\_t* **id**  
Gatt id, include uuid and instance

`bool` **is\_primary**  
This service is primary or not

**struct esp\_attr\_desc\_t**  
Attribute description (used to create database)

### Public Members

`uint16_t uuid_length`  
UUID length

`uint8_t *uuid_p`  
UUID value

`uint16_t perm`  
Attribute permission

`uint16_t max_length`  
Maximum length of the element

`uint16_t length`  
Current length of the element

`uint8_t *value`  
Element value array

**struct esp\_attr\_control\_t**  
attribute auto response flag

### Public Members

`uint8_t auto_rsp`  
if `auto_rsp` set to `ESP_GATT_RSP_BY_APP`, means the response of Write/Read operation will be replied by application. if `auto_rsp` set to `ESP_GATT_AUTO_RSP`, means the response of Write/Read operation will be replied by GATT stack automatically.

**struct esp\_gatts\_attr\_db\_t**  
attribute type added to the gatt server database

### Public Members

*esp\_attr\_control\_t* **attr\_control**  
The attribute control type

*esp\_attr\_desc\_t* **att\_desc**  
The attribute type

**struct esp\_attr\_value\_t**  
set the attribute value type

### Public Members

`uint16_t attr_max_len`  
attribute max value length

`uint16_t attr_len`  
attribute current value length

`uint8_t *attr_value`  
the pointer to attribute value

**struct esp\_gatts\_incl\_svc\_desc\_t**  
Gatt include service entry element.

### Public Members

`uint16_t start_hdl`  
Gatt start handle value of included service

`uint16_t end_hdl`  
Gatt end handle value of included service

`uint16_t uuid`  
Gatt attribute value UUID of included service

**struct esp\_gatts\_incl128\_svc\_desc\_t**  
Gatt include 128 bit service entry element.

### Public Members

`uint16_t start_hdl`  
Gatt start handle value of included 128 bit service

`uint16_t end_hdl`  
Gatt end handle value of included 128 bit service

**struct esp\_gatt\_value\_t**  
Gatt attribute value.

### Public Members

`uint8_t value[ESP_GATT_MAX_ATTR_LEN]`  
Gatt attribute value

`uint16_t handle`  
Gatt attribute handle

`uint16_t offset`  
Gatt attribute value offset

`uint16_t len`  
Gatt attribute value length

`uint8_t auth_req`  
Gatt authentication request

**struct esp\_gattc\_multi\_t**  
read multiple attribute

### Public Members

`uint8_t num_attr`  
The number of the attribute

`uint16_t handles[ESP_GATT_MAX_READ_MULTI_HANDLES]`  
The handles list

**struct esp\_gattc\_db\_elem\_t**  
data base attribute element

## Public Members

*esp\_gatt\_db\_attr\_type\_t* **type**

The attribute type

`uint16_t` **attribute\_handle**

The attribute handle, it's valid for all of the type

`uint16_t` **start\_handle**

The service start handle, it's valid only when the type = ESP\_GATT\_DB\_PRIMARY\_SERVICE or ESP\_GATT\_DB\_SECONDARY\_SERVICE

`uint16_t` **end\_handle**

The service end handle, it's valid only when the type = ESP\_GATT\_DB\_PRIMARY\_SERVICE or ESP\_GATT\_DB\_SECONDARY\_SERVICE

*esp\_gatt\_char\_prop\_t* **properties**

The characteristic properties, it's valid only when the type = ESP\_GATT\_DB\_CHARACTERISTIC

*esp\_bt\_uuid\_t* **uuid**

The attribute uuid, it's valid for all of the type

**struct esp\_gattc\_service\_elem\_t**

service element

## Public Members

bool **is\_primary**

The service flag, ture if the service is primary service, else is secondly service

`uint16_t` **start\_handle**

The start handle of the service

`uint16_t` **end\_handle**

The end handle of the service

*esp\_bt\_uuid\_t* **uuid**

The uuid of the service

**struct esp\_gattc\_char\_elem\_t**

characteristic element

## Public Members

`uint16_t` **char\_handle**

The characteristic handle

*esp\_gatt\_char\_prop\_t* **properties**

The characteristic properties

*esp\_bt\_uuid\_t* **uuid**

The characteristic uuid

**struct esp\_gattc\_descr\_elem\_t**

descriptor element

## Public Members

`uint16_t handle`

The characteristic descriptor handle

`esp_bt_uuid_t uuid`

The characteristic descriptor uuid

`struct esp_gattc_incl_svc_elem_t`

include service element

## Public Members

`uint16_t handle`

The include service current attribute handle

`uint16_t incl_srvc_s_handle`

The start handle of the service which has been included

`esp_bt_uuid_t uuid`

The include service uuid

## Macros

`ESP_GATT_UUID_IMMEDIATE_ALERT_SVC`

All “ESP\_GATT\_UUID\_XXX” is attribute types

`ESP_GATT_UUID_LINK_LOSS_SVC`

`ESP_GATT_UUID_TX_POWER_SVC`

`ESP_GATT_UUID_CURRENT_TIME_SVC`

`ESP_GATT_UUID_REF_TIME_UPDATE_SVC`

`ESP_GATT_UUID_NEXT_DST_CHANGE_SVC`

`ESP_GATT_UUID_GLUCOSE_SVC`

`ESP_GATT_UUID_HEALTH_THERMOM_SVC`

`ESP_GATT_UUID_DEVICE_INFO_SVC`

`ESP_GATT_UUID_HEART_RATE_SVC`

`ESP_GATT_UUID_PHONE_ALERT_STATUS_SVC`

`ESP_GATT_UUID_BATTERY_SERVICE_SVC`

`ESP_GATT_UUID_BLOOD_PRESSURE_SVC`

`ESP_GATT_UUID_ALERT_NTF_SVC`

`ESP_GATT_UUID_HID_SVC`

`ESP_GATT_UUID_SCAN_PARAMETERS_SVC`

`ESP_GATT_UUID_RUNNING_SPEED_CADENCE_SVC`

`ESP_GATT_UUID_CYCLING_SPEED_CADENCE_SVC`

`ESP_GATT_UUID_CYCLING_POWER_SVC`

`ESP_GATT_UUID_LOCATION_AND_NAVIGATION_SVC`

ESP\_GATT\_UUID\_USER\_DATA\_SVC  
ESP\_GATT\_UUID\_WEIGHT\_SCALE\_SVC  
ESP\_GATT\_UUID\_PRI\_SERVICE  
ESP\_GATT\_UUID\_SEC\_SERVICE  
ESP\_GATT\_UUID\_INCLUDE\_SERVICE  
ESP\_GATT\_UUID\_CHAR\_DECLARE  
ESP\_GATT\_UUID\_CHAR\_EXT\_PROP  
ESP\_GATT\_UUID\_CHAR\_DESCRIPTION  
ESP\_GATT\_UUID\_CHAR\_CLIENT\_CONFIG  
ESP\_GATT\_UUID\_CHAR\_SRVR\_CONFIG  
ESP\_GATT\_UUID\_CHAR\_PRESENT\_FORMAT  
ESP\_GATT\_UUID\_CHAR\_AGG\_FORMAT  
ESP\_GATT\_UUID\_CHAR\_VALID\_RANGE  
ESP\_GATT\_UUID\_EXT\_RPT\_REF\_DESCR  
ESP\_GATT\_UUID\_RPT\_REF\_DESCR  
ESP\_GATT\_UUID\_GAP\_DEVICE\_NAME  
ESP\_GATT\_UUID\_GAP\_ICON  
ESP\_GATT\_UUID\_GAP\_PREF\_CONN\_PARAM  
ESP\_GATT\_UUID\_GAP\_CENTRAL\_ADDR\_RESOL  
ESP\_GATT\_UUID\_GATT\_SRV\_CHGD  
ESP\_GATT\_UUID\_ALERT\_LEVEL  
ESP\_GATT\_UUID\_TX\_POWER\_LEVEL  
ESP\_GATT\_UUID\_CURRENT\_TIME  
ESP\_GATT\_UUID\_LOCAL\_TIME\_INFO  
ESP\_GATT\_UUID\_REF\_TIME\_INFO  
ESP\_GATT\_UUID\_NW\_STATUS  
ESP\_GATT\_UUID\_NW\_TRIGGER  
ESP\_GATT\_UUID\_ALERT\_STATUS  
ESP\_GATT\_UUID\_RINGER\_CP  
ESP\_GATT\_UUID\_RINGER\_SETTING  
ESP\_GATT\_UUID\_GM\_MEASUREMENT  
ESP\_GATT\_UUID\_GM\_CONTEXT  
ESP\_GATT\_UUID\_GM\_CONTROL\_POINT  
ESP\_GATT\_UUID\_GM\_FEATURE  
ESP\_GATT\_UUID\_SYSTEM\_ID  
ESP\_GATT\_UUID\_MODEL\_NUMBER\_STR

**ESP\_GATT\_UUID\_SERIAL\_NUMBER\_STR**  
**ESP\_GATT\_UUID\_FW\_VERSION\_STR**  
**ESP\_GATT\_UUID\_HW\_VERSION\_STR**  
**ESP\_GATT\_UUID\_SW\_VERSION\_STR**  
**ESP\_GATT\_UUID\_MANU\_NAME**  
**ESP\_GATT\_UUID\_IEEE\_DATA**  
**ESP\_GATT\_UUID\_PNP\_ID**  
**ESP\_GATT\_UUID\_HID\_INFORMATION**  
**ESP\_GATT\_UUID\_HID\_REPORT\_MAP**  
**ESP\_GATT\_UUID\_HID\_CONTROL\_POINT**  
**ESP\_GATT\_UUID\_HID\_REPORT**  
**ESP\_GATT\_UUID\_HID\_PROTO\_MODE**  
**ESP\_GATT\_UUID\_HID\_BT\_KB\_INPUT**  
**ESP\_GATT\_UUID\_HID\_BT\_KB\_OUTPUT**  
**ESP\_GATT\_UUID\_HID\_BT\_MOUSE\_INPUT**  
**ESP\_GATT\_HEART\_RATE\_MEAS**  
Heart Rate Measurement.  
**ESP\_GATT\_BODY\_SENSOR\_LOCATION**  
Body Sensor Location.  
**ESP\_GATT\_HEART\_RATE\_CNTL\_POINT**  
Heart Rate Control Point.  
**ESP\_GATT\_UUID\_BATTERY\_LEVEL**  
**ESP\_GATT\_UUID\_SC\_CONTROL\_POINT**  
**ESP\_GATT\_UUID\_SENSOR\_LOCATION**  
**ESP\_GATT\_UUID\_RSC\_MEASUREMENT**  
**ESP\_GATT\_UUID\_RSC\_FEATURE**  
**ESP\_GATT\_UUID\_CSC\_MEASUREMENT**  
**ESP\_GATT\_UUID\_CSC\_FEATURE**  
**ESP\_GATT\_UUID\_SCAN\_INT\_WINDOW**  
**ESP\_GATT\_UUID\_SCAN\_REFRESH**  
**ESP\_GATT\_ILLEGAL\_UUID**  
GATT INVALID UUID.  
**ESP\_GATT\_ILLEGAL\_HANDLE**  
GATT INVALID HANDLE.  
**ESP\_GATT\_ATTR\_HANDLE\_MAX**  
GATT attribute max handle.  
**ESP\_GATT\_MAX\_READ\_MULTI\_HANDLES**

**ESP\_GATT\_PERM\_READ**

Attribute permissions.

**ESP\_GATT\_PERM\_READ\_ENCRYPTED**

**ESP\_GATT\_PERM\_READ\_ENC\_MITM**

**ESP\_GATT\_PERM\_WRITE**

**ESP\_GATT\_PERM\_WRITE\_ENCRYPTED**

**ESP\_GATT\_PERM\_WRITE\_ENC\_MITM**

**ESP\_GATT\_PERM\_WRITE\_SIGNED**

**ESP\_GATT\_PERM\_WRITE\_SIGNED\_MITM**

**ESP\_GATT\_CHAR\_PROP\_BIT\_BROADCAST**

**ESP\_GATT\_CHAR\_PROP\_BIT\_READ**

**ESP\_GATT\_CHAR\_PROP\_BIT\_WRITE\_NR**

**ESP\_GATT\_CHAR\_PROP\_BIT\_WRITE**

**ESP\_GATT\_CHAR\_PROP\_BIT\_NOTIFY**

**ESP\_GATT\_CHAR\_PROP\_BIT\_INDICATE**

**ESP\_GATT\_CHAR\_PROP\_BIT\_AUTH**

**ESP\_GATT\_CHAR\_PROP\_BIT\_EXT\_PROP**

**ESP\_GATT\_MAX\_ATTR\_LEN**

GATT maximum attribute length.

**ESP\_GATT\_RSP\_BY\_APP**

**ESP\_GATT\_AUTO\_RSP**

**ESP\_GATT\_IF\_NONE**

If callback report `gattc_if/gatts_if` as this macro, means this event is not correspond to any app

## Type Definitions

```
typedef uint16_t esp_gatt_perm_t
```

```
typedef uint8_t esp_gatt_char_prop_t
```

```
typedef uint8_t esp_gatt_if_t
```

Gatt interface type, different application on GATT client use different `gatt_if`

## Enumerations

```
enum esp_gatt_prep_write_type
```

Attribute write data type from the client.

*Values:*

```
ESP_GATT_PREP_WRITE_CANCEL = 0x00
```

Prepare write cancel

```
ESP_GATT_PREP_WRITE_EXEC = 0x01
```

Prepare write execute

**enum esp\_gatt\_status\_t**

GATT success code and error codes.

*Values:*

```
ESP_GATT_OK = 0x0
ESP_GATT_INVALID_HANDLE = 0x01
ESP_GATT_READ_NOT_PERMIT = 0x02
ESP_GATT_WRITE_NOT_PERMIT = 0x03
ESP_GATT_INVALID_PDU = 0x04
ESP_GATT_INSUF_AUTHENTICATION = 0x05
ESP_GATT_REQ_NOT_SUPPORTED = 0x06
ESP_GATT_INVALID_OFFSET = 0x07
ESP_GATT_INSUF_AUTHORIZATION = 0x08
ESP_GATT_PREPARE_Q_FULL = 0x09
ESP_GATT_NOT_FOUND = 0x0a
ESP_GATT_NOT_LONG = 0x0b
ESP_GATT_INSUF_KEY_SIZE = 0x0c
ESP_GATT_INVALID_ATTR_LEN = 0x0d
ESP_GATT_ERR_UNLIKELY = 0x0e
ESP_GATT_INSUF_ENCRYPTION = 0x0f
ESP_GATT_UNSUPPORT_GRP_TYPE = 0x10
ESP_GATT_INSUF_RESOURCE = 0x11
ESP_GATT_NO_RESOURCES = 0x80
ESP_GATT_INTERNAL_ERROR = 0x81
ESP_GATT_WRONG_STATE = 0x82
ESP_GATT_DB_FULL = 0x83
ESP_GATT_BUSY = 0x84
ESP_GATT_ERROR = 0x85
ESP_GATT_CMD_STARTED = 0x86
ESP_GATT_ILLEGAL_PARAMETER = 0x87
ESP_GATT_PENDING = 0x88
ESP_GATT_AUTH_FAIL = 0x89
ESP_GATT_MORE = 0x8a
ESP_GATT_INVALID_CFG = 0x8b
ESP_GATT_SERVICE_STARTED = 0x8c
ESP_GATT_ENCRYPED_MITM = ESP_GATT_OK
ESP_GATT_ENCRYPED_NO_MITM = 0x8d
```

**ESP\_GATT\_NOT\_ENCRYPTED** = 0x8e  
**ESP\_GATT\_CONGESTED** = 0x8f  
**ESP\_GATT\_DUP\_REG** = 0x90  
**ESP\_GATT\_ALREADY\_OPEN** = 0x91  
**ESP\_GATT\_CANCEL** = 0x92  
**ESP\_GATT\_STACK\_RSP** = 0xe0  
**ESP\_GATT\_APP\_RSP** = 0xe1  
**ESP\_GATT\_UNKNOWN\_ERROR** = 0xef  
**ESP\_GATT\_CCC\_CFG\_ERR** = 0xfd  
**ESP\_GATT\_PRC\_IN\_PROGRESS** = 0xfe  
**ESP\_GATT\_OUT\_OF\_RANGE** = 0xff

**enum esp\_gatt\_conn\_reason\_t**

Gatt Connection reason enum.

*Values:*

**ESP\_GATT\_CONN\_UNKNOWN** = 0  
Gatt connection unknown  
**ESP\_GATT\_CONN\_L2C\_FAILURE** = 1  
General L2cap failure  
**ESP\_GATT\_CONN\_TIMEOUT** = 0x08  
Connection timeout  
**ESP\_GATT\_CONN\_TERMINATE\_PEER\_USER** = 0x13  
Connection terminate by peer user  
**ESP\_GATT\_CONN\_TERMINATE\_LOCAL\_HOST** = 0x16  
Connection terminated by local host  
**ESP\_GATT\_CONN\_FAIL\_ESTABLISH** = 0x3e  
Connection fail to establish  
**ESP\_GATT\_CONN\_LMP\_TIMEOUT** = 0x22  
Connection fail for LMP response tout  
**ESP\_GATT\_CONN\_CONN\_CANCEL** = 0x0100  
L2CAP connection cancelled  
**ESP\_GATT\_CONN\_NONE** = 0x0101  
No connection to cancel

**enum esp\_gatt\_auth\_req\_t**

Gatt authentication request type.

*Values:*

**ESP\_GATT\_AUTH\_REQ\_NONE** = 0  
**ESP\_GATT\_AUTH\_REQ\_NO\_MITM** = 1  
**ESP\_GATT\_AUTH\_REQ\_MITM** = 2  
**ESP\_GATT\_AUTH\_REQ\_SIGNED\_NO\_MITM** = 3  
**ESP\_GATT\_AUTH\_REQ\_SIGNED\_MITM** = 4

**enum esp\_gatt\_write\_type\_t**

Gatt write type.

*Values:*

**ESP\_GATT\_WRITE\_TYPE\_NO\_RSP = 1**

Gatt write attribute need no response

**ESP\_GATT\_WRITE\_TYPE\_RSP**

Gatt write attribute need remote response

**enum esp\_gatt\_db\_attr\_type\_t**

the type of attribute element

*Values:*

**ESP\_GATT\_DB\_PRIMARY\_SERVICE**

Gattc primary service attribute type in the cache

**ESP\_GATT\_DB\_SECONDARY\_SERVICE**

Gattc secondary service attribute type in the cache

**ESP\_GATT\_DB\_CHARACTERISTIC**

Gattc characteristic attribute type in the cache

**ESP\_GATT\_DB\_DESCRIPTOR**

Gattc characteristic descriptor attribute type in the cache

**ESP\_GATT\_DB\_INCLUDED\_SERVICE**

Gattc include service attribute type in the cache

**ESP\_GATT\_DB\_ALL**

Gattc all the attribute (primary service & secondary service & include service & char & descriptor) type in the cache

## GATT SERVER API

### Overview

[Instructions](#)

### Application Example

Check [bluetooth](#) folder in ESP-IDF examples, which contains the following application:

- This is a GATT server demo. Use GATT API to create a GATT server with send advertising. This GATT server can be connected and the service can be discovery - [bluetooth/gatt\\_server](#)

### API Reference

#### Header File

- [bt/bluedroid/api/include/esp\\_gatts\\_api.h](#)

## Functions

esp\_err\_t **esp\_ble\_gatts\_register\_callback** (*esp\_gatts\_cb\_t* callback)

This function is called to register application callbacks with BTA GATTS module.

### Return

- ESP\_OK : success
- other : failed

esp\_err\_t **esp\_ble\_gatts\_app\_register** (uint16\_t *app\_id*)

This function is called to register application identifier.

### Return

- ESP\_OK : success
- other : failed

esp\_err\_t **esp\_ble\_gatts\_app\_unregister** (*esp\_gatt\_if\_t* *gatts\_if*)

unregister with GATT Server.

### Return

- ESP\_OK : success
- other : failed

### Parameters

- *gatts\_if*: GATT server access interface

esp\_err\_t **esp\_ble\_gatts\_create\_service** (*esp\_gatt\_if\_t* *gatts\_if*, *esp\_gatt\_srv\_id\_t* \**service\_id*,  
uint16\_t *num\_handle*)

Create a service. When service creation is done, a callback event BTA\_GATTS\_CREATE\_SRVC\_EVT is called to report status and service ID to the profile. The service ID obtained in the callback function needs to be used when adding included service and characteristics/descriptors into the service.

### Return

- ESP\_OK : success
- other : failed

### Parameters

- *gatts\_if*: GATT server access interface
- *service\_id*: service ID.
- *num\_handle*: number of handle requested for this service.

esp\_err\_t **esp\_ble\_gatts\_create\_attr\_tab** (const *esp\_gatts\_attr\_db\_t* \**gatts\_attr\_db*,  
*esp\_gatt\_if\_t* *gatts\_if*, uint8\_t *max\_nb\_attr*, uint8\_t  
*srvc\_inst\_id*)

Create a service attribute tab.

### Return

- ESP\_OK : success

- other : failed

#### Parameters

- gatts\_attr\_db: the pointer to the service attr tab
- gatts\_if: GATT server access interface
- max\_nb\_attr: the number of attribute to be added to the service database.
- srvc\_inst\_id: the instance id of the service

esp\_err\_t **esp\_ble\_gatts\_add\_included\_service** (uint16\_t *service\_handle*, uint16\_t *included\_service\_handle*)

This function is called to add an included service. After included service is included, a callback event BTA\_GATTS\_ADD\_INCL\_SRVC\_EVT is reported the included service ID.

#### Return

- ESP\_OK : success
- other : failed

#### Parameters

- service\_handle: service handle to which this included service is to be added.
- included\_service\_handle: the service ID to be included.

esp\_err\_t **esp\_ble\_gatts\_add\_char** (uint16\_t *service\_handle*, esp\_bt\_uuid\_t *\*char\_uuid*, esp\_gatt\_perm\_t *perm*, esp\_gatt\_char\_prop\_t *property*, esp\_attr\_value\_t *\*char\_val*, esp\_attr\_control\_t *\*control*)

This function is called to add a characteristic into a service.

#### Return

- ESP\_OK : success
- other : failed

#### Parameters

- service\_handle: service handle to which this included service is to be added.
- char\_uuid: : Characteristic UUID.
- perm: : Characteristic value declaration attribute permission.
- property: : Characteristic Properties
- char\_val: : Characteristic value
- control: : attribute response control byte

esp\_err\_t **esp\_ble\_gatts\_add\_char\_descr** (uint16\_t *service\_handle*, esp\_bt\_uuid\_t *\*descr\_uuid*, esp\_gatt\_perm\_t *perm*, esp\_attr\_value\_t *\*char\_descr\_val*, esp\_attr\_control\_t *\*control*)

This function is called to add characteristic descriptor. When it's done, a callback event BTA\_GATTS\_ADD\_DESCR\_EVT is called to report the status and an ID number for this descriptor.

#### Return

- ESP\_OK : success
- other : failed

### Parameters

- `service_handle`: service handle to which this characteristic descriptor is to be added.
- `perm`: descriptor access permission.
- `descr_uuid`: descriptor UUID.
- `char_descr_val`: : Characteristic descriptor value
- `control`: : attribute response control byte

`esp_err_t esp_ble_gatts_delete_service` (`uint16_t service_handle`)

This function is called to delete a service. When this is done, a callback event `BTA_GATTS_DELETE_EVT` is report with the status.

### Return

- `ESP_OK` : success
- `other` : failed

### Parameters

- `service_handle`: `service_handle` to be deleted.

`esp_err_t esp_ble_gatts_start_service` (`uint16_t service_handle`)

This function is called to start a service.

### Return

- `ESP_OK` : success
- `other` : failed

### Parameters

- `service_handle`: the service handle to be started.

`esp_err_t esp_ble_gatts_stop_service` (`uint16_t service_handle`)

This function is called to stop a service.

### Return

- `ESP_OK` : success
- `other` : failed

### Parameters

- `service_handle`: - service to be topped.

`esp_err_t esp_ble_gatts_send_indicate` (`esp_gatt_if_t gatts_if`, `uint16_t conn_id`, `uint16_t attr_handle`, `uint16_t value_len`, `uint8_t *value`, `bool need_confirm`)

Send indicate or notify to GATT client. Set param `need_confirm` as false will send notification, otherwise indication.

### Return

- `ESP_OK` : success
- `other` : failed

**Parameters**

- `gatts_if`: GATT server access interface
- `conn_id`: - connection id to indicate.
- `attr_handle`: - attribute handle to indicate.
- `value_len`: - indicate value length.
- `value`: value to indicate.
- `need_confirm`: - Whether a confirmation is required. false sends a GATT notification, true sends a GATT indication.

`esp_err_t esp_ble_gatts_send_response` (*esp\_gatt\_if\_t gatts\_if*, `uint16_t conn_id`, `uint32_t trans_id`, *esp\_gatt\_status\_t status*, *esp\_gatt\_rsp\_t \*rsp*)

This function is called to send a response to a request.

**Return**

- `ESP_OK` : success
- other : failed

**Parameters**

- `gatts_if`: GATT server access interface
- `conn_id`: - connection identifier.
- `trans_id`: - transfer id
- `status`: - response status
- `rsp`: - response data.

`esp_err_t esp_ble_gatts_set_attr_value` (`uint16_t attr_handle`, `uint16_t length`, `const uint8_t *value`)

This function is called to set the attribute value by the application.

**Return**

- `ESP_OK` : success
- other : failed

**Parameters**

- `attr_handle`: the attribute handle which to be set
- `length`: the value length
- `value`: the pointer to the attribute value

*esp\_gatt\_status\_t* `esp_ble_gatts_get_attr_value` (`uint16_t attr_handle`, `uint16_t *length`, `const uint8_t **value`)

Retrieve attribute value.

**Return**

- `ESP_GATT_OK` : success
- other : failed

**Parameters**

- `attr_handle`: Attribute handle.
- `length`: pointer to the attribute value length
- `value`: Pointer to attribute value payload, the value cannot be modified by user

`esp_err_t esp_ble_gatts_open` (`esp_gatt_if_t gatts_if`, `esp_bd_addr_t remote_bda`, `bool is_direct`)

Open a direct open connection or add a background auto connection.

#### Return

- `ESP_OK` : success
- other : failed

#### Parameters

- `gatts_if`: GATT server access interface
- `remote_bda`: remote device bluetooth device address.
- `is_direct`: direct connection or background auto connection

`esp_err_t esp_ble_gatts_close` (`esp_gatt_if_t gatts_if`, `uint16_t conn_id`)

Close a connection a remote device.

#### Return

- `ESP_OK` : success
- other : failed

#### Parameters

- `gatts_if`: GATT server access interface
- `conn_id`: connection ID to be closed.

## Unions

`union esp_ble_gatts_cb_param_t`

`#include <esp_gatts_api.h>` Gatt server callback parameters union.

#### Public Members

`struct esp_ble_gatts_cb_param_t::gatts_reg_evt_param` **reg**

Gatt server callback param of `ESP_GATTS_REG_EVT`

`struct esp_ble_gatts_cb_param_t::gatts_read_evt_param` **read**

Gatt server callback param of `ESP_GATTS_READ_EVT`

`struct esp_ble_gatts_cb_param_t::gatts_write_evt_param` **write**

Gatt server callback param of `ESP_GATTS_WRITE_EVT`

`struct esp_ble_gatts_cb_param_t::gatts_exec_write_evt_param` **exec\_write**

Gatt server callback param of `ESP_GATTS_EXEC_WRITE_EVT`

`struct esp_ble_gatts_cb_param_t::gatts_mtu_evt_param` **mtu**

Gatt server callback param of `ESP_GATTS_MTU_EVT`

```

struct esp_ble_gatts_cb_param_t::gatts_conf_evt_param conf
    Gatt server callback param of ESP_GATTS_CONF_EVT (confirm)

struct esp_ble_gatts_cb_param_t::gatts_create_evt_param create
    Gatt server callback param of ESP_GATTS_CREATE_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_incl_srvc_evt_param add_incl_srvc
    Gatt server callback param of ESP_GATTS_ADD_INCL_SRVC_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_char_evt_param add_char
    Gatt server callback param of ESP_GATTS_ADD_CHAR_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_char_descr_evt_param add_char_descr
    Gatt server callback param of ESP_GATTS_ADD_CHAR_DESCR_EVT

struct esp_ble_gatts_cb_param_t::gatts_delete_evt_param del
    Gatt server callback param of ESP_GATTS_DELETE_EVT

struct esp_ble_gatts_cb_param_t::gatts_start_evt_param start
    Gatt server callback param of ESP_GATTS_START_EVT

struct esp_ble_gatts_cb_param_t::gatts_stop_evt_param stop
    Gatt server callback param of ESP_GATTS_STOP_EVT

struct esp_ble_gatts_cb_param_t::gatts_connect_evt_param connect
    Gatt server callback param of ESP_GATTS_CONNECT_EVT

struct esp_ble_gatts_cb_param_t::gatts_disconnect_evt_param disconnect
    Gatt server callback param of ESP_GATTS_DISCONNECT_EVT

struct esp_ble_gatts_cb_param_t::gatts_open_evt_param open
    Gatt server callback param of ESP_GATTS_OPEN_EVT

struct esp_ble_gatts_cb_param_t::gatts_cancel_open_evt_param cancel_open
    Gatt server callback param of ESP_GATTS_CANCEL_OPEN_EVT

struct esp_ble_gatts_cb_param_t::gatts_close_evt_param close
    Gatt server callback param of ESP_GATTS_CLOSE_EVT

struct esp_ble_gatts_cb_param_t::gatts_congest_evt_param congest
    Gatt server callback param of ESP_GATTS_CONGEST_EVT

struct esp_ble_gatts_cb_param_t::gatts_rsp_evt_param rsp
    Gatt server callback param of ESP_GATTS_RESPONSE_EVT

struct esp_ble_gatts_cb_param_t::gatts_add_attr_tab_evt_param add_attr_tab
    Gatt server callback param of ESP_GATTS_CREAT_ATTR_TAB_EVT

struct esp_ble_gatts_cb_param_t::gatts_set_attr_val_evt_param set_attr_val
    Gatt server callback param of ESP_GATTS_SET_ATTR_VAL_EVT

struct gatts_add_attr_tab_evt_param
    #include <esp_gatts_api.h> ESP_GATTS_CREAT_ATTR_TAB_EVT.

```

## Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

*esp\_bt\_uuid\_t* **svc\_uuid**  
Service uuid type

**uint16\_t num\_handle**  
The number of the attribute handle to be added to the gatts database

**uint16\_t \*handles**  
The number to the handles

**struct gatts\_add\_char\_descr\_evt\_param**  
*#include <esp\_gatts\_api.h> ESP\_GATTS\_ADD\_CHAR\_DESCR\_EVT.*

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**uint16\_t attr\_handle**  
Descriptor attribute handle

**uint16\_t service\_handle**  
Service attribute handle

*esp\_bt\_uuid\_t* **descr\_uuid**  
Characteristic descriptor uuid

**struct gatts\_add\_char\_evt\_param**  
*#include <esp\_gatts\_api.h> ESP\_GATTS\_ADD\_CHAR\_EVT.*

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**uint16\_t attr\_handle**  
Characteristic attribute handle

**uint16\_t service\_handle**  
Service attribute handle

*esp\_bt\_uuid\_t* **char\_uuid**  
Characteristic uuid

**struct gatts\_add\_incl\_srvc\_evt\_param**  
*#include <esp\_gatts\_api.h> ESP\_GATTS\_ADD\_INCL\_SRVC\_EVT.*

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**uint16\_t attr\_handle**  
Included service attribute handle

**uint16\_t service\_handle**  
Service attribute handle

**struct gatts\_cancel\_open\_evt\_param**  
*#include <esp\_gatts\_api.h> ESP\_GATTS\_CANCEL\_OPEN\_EVT.*

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**struct gatts\_close\_evt\_param**  
*#include <esp\_gatts\_api.h>* ESP\_GATTS\_CLOSE\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

uint16\_t **conn\_id**  
Connection id

**struct gatts\_conf\_evt\_param**  
*#include <esp\_gatts\_api.h>* ESP\_GATTS\_CONF\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

uint16\_t **conn\_id**  
Connection id

uint16\_t **len**  
The indication or notification value length, len is valid when send notification or indication failed

uint8\_t **\*value**  
The indication or notification value , value is valid when send notification or indication failed

**struct gatts\_congest\_evt\_param**  
*#include <esp\_gatts\_api.h>* ESP\_GATTS\_LISTEN\_EVT.  
ESP\_GATTS\_CONGEST\_EVT

### Public Members

uint16\_t **conn\_id**  
Connection id

bool **congested**  
Congested or not

**struct gatts\_connect\_evt\_param**  
*#include <esp\_gatts\_api.h>* ESP\_GATTS\_CONNECT\_EVT.

### Public Members

uint16\_t **conn\_id**  
Connection id

*esp\_bd\_addr\_t* **remote\_bda**  
Remote bluetooth device address

```
struct gatts_create_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_UNREG_EVT.  
    ESP_GATTS_CREATE_EVT
```

### Public Members

```
esp_gatt_status_t status  
    Operation status  
  
uint16_t service_handle  
    Service attribute handle  
  
esp_gatt_srv_id_t service_id  
    Service id, include service uuid and other information
```

```
struct gatts_delete_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_DELETE_EVT.
```

### Public Members

```
esp_gatt_status_t status  
    Operation status  
  
uint16_t service_handle  
    Service attribute handle  
  
struct gatts_disconnect_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_DISCONNECT_EVT.
```

### Public Members

```
uint16_t conn_id  
    Connection id  
  
esp_bd_addr_t remote_bda  
    Remote bluetooth device address  
  
esp_gatt_conn_reason_t reason  
    Indicate the reason of disconnection
```

```
struct gatts_exec_write_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_EXEC_WRITE_EVT.
```

### Public Members

```
uint16_t conn_id  
    Connection id  
  
uint32_t trans_id  
    Transfer id  
  
esp_bd_addr_t bda  
    The bluetooth device address which been written  
  
uint8_t exec_write_flag  
    Execute write flag
```

```
struct gatts_mtu_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_MTU_EVT.
```

### Public Members

uint16\_t **conn\_id**  
Connection id

uint16\_t **mtu**  
MTU size

```
struct gatts_open_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_OPEN_EVT.
```

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

```
struct gatts_read_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_READ_EVT.
```

### Public Members

uint16\_t **conn\_id**  
Connection id

uint32\_t **trans\_id**  
Transfer id

*esp\_bd\_addr\_t* **bda**  
The bluetooth device address which been read

uint16\_t **handle**  
The attribute handle

uint16\_t **offset**  
Offset of the value, if the value is too long

bool **is\_long**  
The value is too long or not

bool **need\_rsp**  
The read operation need to do response

```
struct gatts_reg_evt_param  
#include <esp_gatts_api.h> ESP_GATTS_REG_EVT.
```

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

uint16\_t **app\_id**  
Application id which input in register API

```
struct gatts_rsp_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_RESPONSE_EVT.
```

### Public Members

*esp\_gatt\_status\_t* **status**  
 Operation status

uint16\_t **handle**  
 Attribute handle which send response

```
struct gatts_set_attr_val_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_SET_ATTR_VAL_EVT.
```

### Public Members

uint16\_t **srvc\_handle**  
 The service handle

uint16\_t **attr\_handle**  
 The attribute handle

*esp\_gatt\_status\_t* **status**  
 Operation status

```
struct gatts_start_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_START_EVT.
```

### Public Members

*esp\_gatt\_status\_t* **status**  
 Operation status

uint16\_t **service\_handle**  
 Service attribute handle

```
struct gatts_stop_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_STOP_EVT.
```

### Public Members

*esp\_gatt\_status\_t* **status**  
 Operation status

uint16\_t **service\_handle**  
 Service attribute handle

```
struct gatts_write_evt_param  
    #include <esp_gatts_api.h> ESP_GATTS_WRITE_EVT.
```

## Public Members

`uint16_t conn_id`  
Connection id

`uint32_t trans_id`  
Transfer id

`esp_bd_addr_t bda`  
The bluetooth device address which been written

`uint16_t handle`  
The attribute handle

`uint16_t offset`  
Offset of the value, if the value is too long

`bool need_rsp`  
The write operation need to do response

`bool is_prep`  
This write operation is prepare write

`uint16_t len`  
The write attribute value length

`uint8_t *value`  
The write attribute value

## Macros

**ESP\_GATT\_PREP\_WRITE\_CANCEL**  
Prepare write flag to indicate cancel prepare write

**ESP\_GATT\_PREP\_WRITE\_EXEC**  
Prepare write flag to indicate execute prepare write

## Type Definitions

```
typedef void (*esp_gatts_cb_t) (esp_gatts_cb_event_t event, esp_gatt_if_t gatts_if,
                               esp_ble_gatts_cb_param_t *param)
```

GATT Server callback function type.

### Parameters

- `event`: : Event type
- `gatts_if`: : GATT server access interface, normally different `gatts_if` correspond to different profile
- `param`: : Point to callback parameter, currently is union type

## Enumerations

**enum esp\_gatts\_cb\_event\_t**  
GATT Server callback function events.

*Values:*

**ESP\_GATTS\_REG\_EVT = 0**  
When register application id, the event comes

**ESP\_GATTS\_READ\_EVT = 1**  
When gatt client request read operation, the event comes

**ESP\_GATTS\_WRITE\_EVT = 2**  
When gatt client request write operation, the event comes

**ESP\_GATTS\_EXEC\_WRITE\_EVT = 3**  
When gatt client request execute write, the event comes

**ESP\_GATTS\_MTU\_EVT = 4**  
When set mtu complete, the event comes

**ESP\_GATTS\_CONF\_EVT = 5**  
When receive confirm, the event comes

**ESP\_GATTS\_UNREG\_EVT = 6**  
When unregister application id, the event comes

**ESP\_GATTS\_CREATE\_EVT = 7**  
When create service complete, the event comes

**ESP\_GATTS\_ADD\_INCL\_SRVC\_EVT = 8**  
When add included service complete, the event comes

**ESP\_GATTS\_ADD\_CHAR\_EVT = 9**  
When add characteristic complete, the event comes

**ESP\_GATTS\_ADD\_CHAR\_DESCR\_EVT = 10**  
When add descriptor complete, the event comes

**ESP\_GATTS\_DELETE\_EVT = 11**  
When delete service complete, the event comes

**ESP\_GATTS\_START\_EVT = 12**  
When start service complete, the event comes

**ESP\_GATTS\_STOP\_EVT = 13**  
When stop service complete, the event comes

**ESP\_GATTS\_CONNECT\_EVT = 14**  
When gatt client connect, the event comes

**ESP\_GATTS\_DISCONNECT\_EVT = 15**  
When gatt client disconnect, the event comes

**ESP\_GATTS\_OPEN\_EVT = 16**  
When connect to peer, the event comes

**ESP\_GATTS\_CANCEL\_OPEN\_EVT = 17**  
When disconnect from peer, the event comes

**ESP\_GATTS\_CLOSE\_EVT = 18**  
When gatt server close, the event comes

**ESP\_GATTS\_LISTEN\_EVT = 19**  
When gatt listen to be connected the event comes

**ESP\_GATTS\_CONGEST\_EVT = 20**  
When congest happen, the event comes

**ESP\_GATTS\_RESPONSE\_EVT = 21**

When gatt send response complete, the event comes

**ESP\_GATTS\_CREAT\_ATTR\_TAB\_EVT = 22**

**ESP\_GATTS\_SET\_ATTR\_VAL\_EVT = 23**

## GATT CLIENT API

### Overview

Instructions

### Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following application:

- This is a GATT client demo. This demo can scan devices, connect to the GATT server and discover the service `bluetooth/gatt_client`

### API Reference

#### Header File

- `bt/bluedroid/api/include/esp_gattc_api.h`

#### Functions

`esp_err_t esp_ble_gattc_register_callback` (*esp\_gattc\_cb\_t callback*)

This function is called to register application callbacks with GATTC module.

##### Return

- `ESP_OK`: success
- other: failed

##### Parameters

- `callback`: : pointer to the application callback function.

`esp_err_t esp_ble_gattc_app_register` (`uint16_t app_id`)

This function is called to register application callbacks with GATTC module.

##### Return

- `ESP_OK`: success
- other: failed

##### Parameters

- `app_id`: : Application Identify (UUID), for different application

esp\_err\_t **esp\_ble\_gattc\_app\_unregister** (*esp\_gatt\_if\_t gattc\_if*)

This function is called to unregister an application from GATT module.

**Return**

- ESP\_OK: success
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.

esp\_err\_t **esp\_ble\_gattc\_open** (*esp\_gatt\_if\_t gattc\_if, esp\_bd\_addr\_t remote\_bda, esp\_ble\_addr\_type\_t remote\_addr\_type, bool is\_direct*)

Open a direct connection or add a background auto connection.

**Return**

- ESP\_OK: success
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.
- remote\_bda: remote device bluetooth device address.
- remote\_addr\_type: remote device bluetooth device the address type.
- is\_direct: direct connection or background auto connection

esp\_err\_t **esp\_ble\_gattc\_close** (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id*)

Close a virtual connection to a GATT server. gattc maybe have multiple virtual GATT server connections when multiple app\_id registered, this API only close one virtual GATT server connection. if there exist other virtual GATT server connections, it does not disconnect the physical connection. if you want to disconnect the physical connection directly, you can use esp\_ble\_gap\_disconnect(esp\_bd\_addr\_t remote\_device).

**Return**

- ESP\_OK: success
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.
- conn\_id: connection ID to be closed.

esp\_err\_t **esp\_ble\_gattc\_send\_mtu\_req** (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id*)

Configure the MTU size in the GATT channel. This can be done only once per connection. Before using, use esp\_ble\_gatt\_set\_local\_mtu() to configure the local MTU size.

**Return**

- ESP\_OK: success
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.

- `conn_id`: connection ID.

`esp_err_t esp_ble_gattc_search_service` (`esp_gatt_if_t gattc_if`, `uint16_t conn_id`, `esp_bt_uuid_t *filter_uuid`)

This function is called to request a GATT service discovery on a GATT server. This function report service search result by a callback event, and followed by a service search complete event.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID.
- `filter_uuid`: a UUID of the service application is interested in. If Null, discover for all services.

`esp_gatt_status_t esp_ble_gattc_get_service` (`esp_gatt_if_t gattc_if`, `uint16_t conn_id`, `esp_bt_uuid_t *svc_uuid`, `esp_gattc_service_elem_t *result`, `uint16_t *count`, `uint16_t offset`)

Find all the service with the given service uuid in the gattc cache, if the `svc_uuid` is NULL, find all the service. Note: It just get service from local cache, won't get from remote devices. If want to get it from remote device, need to used the `esp_ble_gattc_search_service`.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `svc_uuid`: the pointer to the service uuid.
- `result`: The pointer to the service which has been found in the gattc cache.
- `count`: input the number of service want to find, it will output the number of service has been found in the gattc cache with the given service uuid.
- `offset`: Offset of the service position to get.

`esp_gatt_status_t esp_ble_gattc_get_all_char` (`esp_gatt_if_t gattc_if`, `uint16_t conn_id`, `uint16_t start_handle`, `uint16_t end_handle`, `esp_gattc_char_elem_t *result`, `uint16_t *count`, `uint16_t offset`)

Find all the characteristic with the given service in the gattc cache Note: It just get characteristic from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `start_handle`: the attribute start handle.
- `end_handle`: the attribute end handle
- `result`: The pointer to the characteristic in the service.
- `count`: input the number of characteristic want to find, it will output the number of characteristic has been found in the gattc cache with the given service.
- `offset`: Offset of the characteristic position to get.

`esp_gatt_status_t esp_ble_gattc_get_all_descr(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t char_handle, esp_gattc_descr_elem_t *result, uint16_t *count, uint16_t offset)`

Find all the descriptor with the given characteristic in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `char_handle`: the given characteristic handle
- `result`: The pointer to the descriptor in the characteristic.
- `count`: input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.
- `offset`: Offset of the descriptor position to get.

`esp_gatt_status_t esp_ble_gattc_get_char_by_uuid(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t start_handle, uint16_t end_handle, esp_bt_uuid_t char_uuid, esp_gattc_char_elem_t *result, uint16_t *count)`

Find the characteristic with the given characteristic uuid in the gattc cache Note: It just get characteristic from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `start_handle`: the attribute start handle
- `end_handle`: the attribute end handle

- `char_uuid`: the characteristic uuid
- `result`: The pointer to the characteristic in the service.
- `count`: input the number of characteristic want to find, it will output the number of characteristic has been found in the gattc cache with the given service.

```
esp_gatt_status_t esp_ble_gattc_get_descr_by_uuid(esp_gatt_if_t gattc_if, uint16_t conn_id,
                                                    uint16_t start_handle, uint16_t end_handle,
                                                    esp_bt_uuid_t char_uuid, esp_bt_uuid_t descr_uuid,
                                                    esp_gattc_descr_elem_t *result,
                                                    uint16_t *count)
```

Find the descriptor with the given characteristic uuid in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `start_handle`: the attribute start handle
- `end_handle`: the attribute end handle
- `char_uuid`: the characteristic uuid.
- `descr_uuid`: the descriptor uuid.
- `result`: The pointer to the descriptor in the given characteristic.
- `count`: input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.

```
esp_gatt_status_t esp_ble_gattc_get_descr_by_char_handle(esp_gatt_if_t gattc_if, uint16_t
                                                         conn_id, uint16_t char_handle,
                                                         esp_bt_uuid_t descr_uuid,
                                                         esp_gattc_descr_elem_t *re-
                                                         sult, uint16_t *count)
```

Find the descriptor with the given characteristic handle in the gattc cache Note: It just get descriptor from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `char_handle`: the characteristic handle.
- `descr_uuid`: the descriptor uuid.
- `result`: The pointer to the descriptor in the given characteristic.

- `count`: input the number of descriptor want to find, it will output the number of descriptor has been found in the gattc cache with the given characteristic.

```
esp_gatt_status_t esp_ble_gattc_get_include_service (esp_gatt_if_t gattc_if, uint16_t conn_id,  
                                                    uint16_t start_handle, uint16_t  
                                                    end_handle, esp_bt_uuid_t *incl_uuid,  
                                                    esp_gattc_incl_svc_elem_t *result,  
                                                    uint16_t *count)
```

Find the include service with the given service handle in the gattc cache Note: It just get include service from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `start_handle`: the attribute start handle
- `end_handle`: the attribute end handle
- `incl_uuid`: the include service uuid
- `result`: The pointer to the include service in the given service.
- `count`: input the number of include service want to find, it will output the number of include service has been found in the gattc cache with the given service.

```
esp_gatt_status_t esp_ble_gattc_get_attr_count (esp_gatt_if_t gattc_if, uint16_t conn_id,  
                                                    esp_gatt_db_attr_type_t type, uint16_t  
                                                    start_handle, uint16_t end_handle, uint16_t  
                                                    char_handle, uint16_t *count)
```

Find the attribute count with the given service or characteristic in the gattc cache.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: connection ID which identify the server.
- `type`: the attribute type.
- `start_handle`: the attribute start handle, if the type is `ESP_GATT_DB_DESCRIPTOR`, this parameter should be ignore
- `end_handle`: the attribute end handle, if the type is `ESP_GATT_DB_DESCRIPTOR`, this parameter should be ignore
- `char_handle`: the characteristic handle, this parameter valid when the type is `ESP_GATT_DB_DESCRIPTOR`. If the type isn't `ESP_GATT_DB_DESCRIPTOR`, this parameter should be ignore.

- `count`: output the number of attribute has been found in the gattc cache with the given attribute type.

```
esp_gatt_status_t esp_ble_gattc_get_db(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t
start_handle, uint16_t end_handle, esp_gattc_db_elem_t
*db, uint16_t *count)
```

This function is called to get the GATT database. Note: It just get attribute data base from local cache, won't get from remote devices.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `start_handle`: the attribute start handle
- `end_handle`: the attribute end handle
- `conn_id`: connection ID which identify the server.
- `db`: output parameter which will contain the GATT database copy. Caller is responsible for freeing it.
- `count`: number of elements in database.

```
esp_err_t esp_ble_gattc_read_char(esp_gatt_if_t gattc_if, uint16_t conn_id, uint16_t handle,
esp_gatt_auth_req_t auth_req)
```

This function is called to read a service's characteristics of the given characteristic handle.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: : connection ID.
- `handle`: : characteritic handle to read.
- `auth_req`: : authenticate request type

```
esp_err_t esp_ble_gattc_read_multiple(esp_gatt_if_t gattc_if, uint16_t conn_id, esp_gattc_multi_t
*read_multi, esp_gatt_auth_req_t auth_req)
```

This function is called to read multiple characteristic or characteristic descriptors.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: : connection ID.
- `read_multi`: : pointer to the read multiple parameter.

- `auth_req`: : authenticate request type

`esp_err_t esp_ble_gattc_read_char_descr` (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id, uint16\_t handle, esp\_gatt\_auth\_req\_t auth\_req*)

This function is called to read a characteristics descriptor.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: : connection ID.
- `handle`: : descriptor handle to read.
- `auth_req`: : authenticate request type

`esp_err_t esp_ble_gattc_write_char` (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id, uint16\_t handle, uint16\_t value\_len, uint8\_t \*value, esp\_gatt\_write\_type\_t write\_type, esp\_gatt\_auth\_req\_t auth\_req*)

This function is called to write characteristic value.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: : connection ID.
- `handle`: : characteristic handle to write.
- `value_len`: length of the value to be written.
- `value`: : the value to be written.
- `write_type`: : the type of attribute write operation.
- `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_write_char_descr` (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id, uint16\_t handle, uint16\_t value\_len, uint8\_t \*value, esp\_gatt\_write\_type\_t write\_type, esp\_gatt\_auth\_req\_t auth\_req*)

This function is called to write characteristic descriptor value.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.

- `conn_id`: : connection ID
- `handle`: : descriptor handle to write.
- `value_len`: length of the value to be written.
- `value`: : the value to be written.
- `write_type`: : the type of attribute write operation.
- `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_prepare_write` (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id, uint16\_t handle, uint16\_t offset, uint16\_t value\_len, uint8\_t \*value, esp\_gatt\_auth\_req\_t auth\_req*)

This function is called to prepare write a characteristic value.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: : connection ID.
- `handle`: : characteristic handle to prepare write.
- `offset`: : offset of the write value.
- `value_len`: length of the value to be written.
- `value`: : the value to be written.
- `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_prepare_write_char_descr` (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id, uint16\_t handle, uint16\_t offset, uint16\_t value\_len, uint8\_t \*value, esp\_gatt\_auth\_req\_t auth\_req*)

This function is called to prepare write a characteristic descriptor value.

#### Return

- `ESP_OK`: success
- other: failed

#### Parameters

- `gattc_if`: Gatt client access interface.
- `conn_id`: : connection ID.
- `handle`: : characteristic descriptor handle to prepare write.
- `offset`: : offset of the write value.
- `value_len`: length of the value to be written.
- `value`: : the value to be written.
- `auth_req`: : authentication request.

`esp_err_t esp_ble_gattc_execute_write` (*esp\_gatt\_if\_t gattc\_if, uint16\_t conn\_id, bool is\_execute*)  
This function is called to execute write a prepare write sequence.

**Return**

- ESP\_OK: success
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.
- conn\_id: : connection ID.
- is\_execute: : execute or cancel.

`esp_err_t esp_ble_gattc_register_for_notify` (*esp\_gatt\_if\_t gattc\_if, esp\_bd\_addr\_t server\_bda, uint16\_t handle*)

This function is called to register for notification of a service.

**Return**

- ESP\_OK: registration succeeds
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.
- server\_bda: : target GATT server.
- handle: : GATT characteristic handle.

`esp_err_t esp_ble_gattc_unregister_for_notify` (*esp\_gatt\_if\_t gattc\_if, esp\_bd\_addr\_t server\_bda, uint16\_t handle*)

This function is called to de-register for notification of a service.

**Return**

- ESP\_OK: unregister succeeds
- other: failed

**Parameters**

- gattc\_if: Gatt client access interface.
- server\_bda: : target GATT server.
- handle: : GATT characteristic handle.

`esp_err_t esp_ble_gattc_cache_refresh` (*esp\_bd\_addr\_t remote\_bda*)  
Refresh the server cache store in the gattc stack of the remote device.

**Return**

- ESP\_OK: success
- other: failed

**Parameters**

- remote\_bda: remote device BD address.

## Unions

```
union esp_ble_gattc_cb_param_t
```

```
#include <esp_gattc_api.h> Gatt client callback parameters union.
```

### Public Members

```
struct esp_ble_gattc_cb_param_t::gattc_reg_evt_param reg  
Gatt client callback param of ESP_GATTC_REG_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_open_evt_param open  
Gatt client callback param of ESP_GATTC_OPEN_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_close_evt_param close  
Gatt client callback param of ESP_GATTC_CLOSE_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_cfg_mtu_evt_param cfg_mtu  
Gatt client callback param of ESP_GATTC_CFG_MTU_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_search_cmpl_evt_param search_cmpl  
Gatt client callback param of ESP_GATTC_SEARCH_CMPL_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_search_res_evt_param search_res  
Gatt client callback param of ESP_GATTC_SEARCH_RES_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_read_char_evt_param read  
Gatt client callback param of ESP_GATTC_READ_CHAR_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_write_evt_param write  
Gatt client callback param of ESP_GATTC_WRITE_DESCR_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_exec_cmpl_evt_param exec_cmpl  
Gatt client callback param of ESP_GATTC_EXEC_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_notify_evt_param notify  
Gatt client callback param of ESP_GATTC_NOTIFY_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_srvc_chg_evt_param srvc_chg  
Gatt client callback param of ESP_GATTC_SRVC_CHG_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_congest_evt_param congest  
Gatt client callback param of ESP_GATTC_CONGEST_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_reg_for_notify_evt_param reg_for_notify  
Gatt client callback param of ESP_GATTC_REG_FOR_NOTIFY_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_unreg_for_notify_evt_param unreg_for_notify  
Gatt client callback param of ESP_GATTC_UNREG_FOR_NOTIFY_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_connect_evt_param connect  
Gatt client callback param of ESP_GATTC_CONNECT_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_disconnect_evt_param disconnect  
Gatt client callback param of ESP_GATTC_DISCONNECT_EVT
```

```
struct esp_ble_gattc_cb_param_t::gattc_queue_full_evt_param queue_full  
Gatt client callback param of ESP_GATTC_QUEUE_FULL_EVT
```

```
struct gattc_cfg_mtu_evt_param  
#include <esp_gattc_api.h> ESP_GATTC_CFG_MTU_EVT.
```

### Public Members

*esp\_gatt\_status\_t* **status**

Operation status

uint16\_t **conn\_id**

Connection id

uint16\_t **mtu**

MTU size

**struct gattc\_close\_evt\_param**

*#include <esp\_gattc\_api.h>* ESP\_GATTC\_CLOSE\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**

Operation status

uint16\_t **conn\_id**

Connection id

*esp\_bd\_addr\_t* **remote\_bda**

Remote bluetooth device address

*esp\_gatt\_conn\_reason\_t* **reason**

The reason of gatt connection close

**struct gattc\_congest\_evt\_param**

*#include <esp\_gattc\_api.h>* ESP\_GATTC\_CONGEST\_EVT.

### Public Members

uint16\_t **conn\_id**

Connection id

bool **congested**

Congested or not

**struct gattc\_connect\_evt\_param**

*#include <esp\_gattc\_api.h>* ESP\_GATTC\_CONNECT\_EVT.

### Public Members

uint16\_t **conn\_id**

Connection id

*esp\_bd\_addr\_t* **remote\_bda**

Remote bluetooth device address

**struct gattc\_disconnect\_evt\_param**

*#include <esp\_gattc\_api.h>* ESP\_GATTC\_DISCONNECT\_EVT.

### Public Members

*esp\_gatt\_conn\_reason\_t* **reason**  
disconnection reason

uint16\_t **conn\_id**  
Connection id

*esp\_bd\_addr\_t* **remote\_bda**  
Remote bluetooth device address

**struct gattc\_exec\_cmpl\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_EXEC\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

uint16\_t **conn\_id**  
Connection id

**struct gattc\_notify\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_NOTIFY\_EVT.

### Public Members

uint16\_t **conn\_id**  
Connection id

*esp\_bd\_addr\_t* **remote\_bda**  
Remote bluetooth device address

uint16\_t **handle**  
The Characteristic or descriptor handle

uint16\_t **value\_len**  
Notify attribute value

uint8\_t **\*value**  
Notify attribute value

bool **is\_notify**  
True means notify, false means indicate

**struct gattc\_open\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_OPEN\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

uint16\_t **conn\_id**  
Connection id

*esp\_bd\_addr\_t* **remote\_bda**  
Remote bluetooth device address

`uint16_t mtu`  
MTU size

**struct gattc\_queue\_full\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_QUEUE\_FULL\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

`uint16_t conn_id`  
Connection id

`bool is_full`  
The gattc command queue is full or not

**struct gattc\_read\_char\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_READ\_CHAR\_EVT, ESP\_GATTC\_READ\_DESCR\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

`uint16_t conn_id`  
Connection id

`uint16_t handle`  
Characteristic handle

`uint8_t *value`  
Characteristic value

`uint16_t value_len`  
Characteristic value length

**struct gattc\_reg\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_REG\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

`uint16_t app_id`  
Application id which input in register API

**struct gattc\_reg\_for\_notify\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_REG\_FOR\_NOTIFY\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**uint16\_t handle**  
The characteristic or descriptor handle

**struct gattc\_search\_cmpl\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_SEARCH\_CMPL\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**uint16\_t conn\_id**  
Connection id

**struct gattc\_search\_res\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_SEARCH\_RES\_EVT.

### Public Members

**uint16\_t conn\_id**  
Connection id

**uint16\_t start\_handle**  
Service start handle

**uint16\_t end\_handle**  
Service end handle

*esp\_gatt\_id\_t* **srvc\_id**  
Service id, include service uuid and other information

**struct gattc\_srvc\_chg\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_SRVC\_CHG\_EVT.

### Public Members

*esp\_bd\_addr\_t* **remote\_bda**  
Remote bluetooth device address

**struct gattc\_unreg\_for\_notify\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_UNREG\_FOR\_NOTIFY\_EVT.

### Public Members

*esp\_gatt\_status\_t* **status**  
Operation status

**uint16\_t handle**  
The characteristic or descriptor handle

**struct gattc\_write\_evt\_param**  
*#include <esp\_gattc\_api.h>* ESP\_GATTC\_WRITE\_CHAR\_EVT, ESP\_GATTC\_PREP\_WRITE\_EVT,  
ESP\_GATTC\_WRITE\_DESCR\_EVT.

## Public Members

*esp\_gatt\_status\_t* **status**

Operation status

uint16\_t **conn\_id**

Connection id

uint16\_t **handle**

The Characteristic or descriptor handle

uint16\_t **offset**

The prepare write offset, this value is valid only when prepare write

## Type Definitions

```
typedef void (*esp_gattc_cb_t) (esp_gattc_cb_event_t event, esp_gatt_if_t gattc_if,  
                                esp_ble_gattc_cb_param_t *param)
```

GATT Client callback function type.

### Parameters

- `event`: : Event type
- `gatts_if`: : GATT client access interface, normally different `gattc_if` correspond to different profile
- `param`: : Point to callback parameter, currently is union type

## Enumerations

```
enum esp_gattc_cb_event_t
```

GATT Client callback function events.

*Values:*

```
ESP_GATTC_REG_EVT = 0
```

When GATT client is registered, the event comes

```
ESP_GATTC_UNREG_EVT = 1
```

When GATT client is unregistered, the event comes

```
ESP_GATTC_OPEN_EVT = 2
```

When GATT virtual connection is set up, the event comes

```
ESP_GATTC_READ_CHAR_EVT = 3
```

When GATT characteristic is read, the event comes

```
ESP_GATTC_WRITE_CHAR_EVT = 4
```

When GATT characteristic write operation completes, the event comes

```
ESP_GATTC_CLOSE_EVT = 5
```

When GATT virtual connection is closed, the event comes

```
ESP_GATTC_SEARCH_CMPL_EVT = 6
```

When GATT service discovery is completed, the event comes

```
ESP_GATTC_SEARCH_RES_EVT = 7
```

When GATT service discovery result is got, the event comes

- ESP\_GATTC\_READ\_DESCR\_EVT = 8**  
When GATT characteristic descriptor read completes, the event comes
- ESP\_GATTC\_WRITE\_DESCR\_EVT = 9**  
When GATT characteristic descriptor write completes, the event comes
- ESP\_GATTC\_NOTIFY\_EVT = 10**  
When GATT notification or indication arrives, the event comes
- ESP\_GATTC\_PREP\_WRITE\_EVT = 11**  
When GATT prepare-write operation completes, the event comes
- ESP\_GATTC\_EXEC\_EVT = 12**  
When write execution completes, the event comes
- ESP\_GATTC\_ACL\_EVT = 13**  
When ACL connection is up, the event comes
- ESP\_GATTC\_CANCEL\_OPEN\_EVT = 14**  
When GATT client ongoing connection is cancelled, the event comes
- ESP\_GATTC\_SRVC\_CHG\_EVT = 15**  
When “service changed” occurs, the event comes
- ESP\_GATTC\_ENC\_CMPL\_CB\_EVT = 17**  
When encryption procedure completes, the event comes
- ESP\_GATTC\_CFG\_MTU\_EVT = 18**  
When configuration of MTU completes, the event comes
- ESP\_GATTC\_ADV\_DATA\_EVT = 19**  
When advertising of data, the event comes
- ESP\_GATTC\_MULT\_ADV\_ENB\_EVT = 20**  
When multi-advertising is enabled, the event comes
- ESP\_GATTC\_MULT\_ADV\_UPD\_EVT = 21**  
When multi-advertising parameters are updated, the event comes
- ESP\_GATTC\_MULT\_ADV\_DATA\_EVT = 22**  
When multi-advertising data arrives, the event comes
- ESP\_GATTC\_MULT\_ADV\_DIS\_EVT = 23**  
When multi-advertising is disabled, the event comes
- ESP\_GATTC\_CONGEST\_EVT = 24**  
When GATT connection congestion comes, the event comes
- ESP\_GATTC\_BTH\_SCAN\_ENB\_EVT = 25**  
When batch scan is enabled, the event comes
- ESP\_GATTC\_BTH\_SCAN\_CFG\_EVT = 26**  
When batch scan storage is configured, the event comes
- ESP\_GATTC\_BTH\_SCAN\_RD\_EVT = 27**  
When Batch scan read event is reported, the event comes
- ESP\_GATTC\_BTH\_SCAN\_THR\_EVT = 28**  
When Batch scan threshold is set, the event comes
- ESP\_GATTC\_BTH\_SCAN\_PARAM\_EVT = 29**  
When Batch scan parameters are set, the event comes

**ESP\_GATTC\_BTH\_SCAN\_DIS\_EVT = 30**

When Batch scan is disabled, the event comes

**ESP\_GATTC\_SCAN\_FLT\_CFG\_EVT = 31**

When Scan filter configuration completes, the event comes

**ESP\_GATTC\_SCAN\_FLT\_PARAM\_EVT = 32**

When Scan filter parameters are set, the event comes

**ESP\_GATTC\_SCAN\_FLT\_STATUS\_EVT = 33**

When Scan filter status is reported, the event comes

**ESP\_GATTC\_ADV\_VSC\_EVT = 34**

When advertising vendor spec content event is reported, the event comes

**ESP\_GATTC\_REG\_FOR\_NOTIFY\_EVT = 38**

When register for notification of a service completes, the event comes

**ESP\_GATTC\_UNREG\_FOR\_NOTIFY\_EVT = 39**

When unregister for notification of a service completes, the event comes

**ESP\_GATTC\_CONNECT\_EVT = 40**

When the ble physical connection is set up, the event comes

**ESP\_GATTC\_DISCONNECT\_EVT = 41**

When the ble physical connection disconnected, the event comes

**ESP\_GATTC\_READ\_MULTIPLE\_EVT = 42**

When the ble characteristic or descriptor multiple complete, the event comes

**ESP\_GATTC\_QUEUE\_FULL\_EVT = 43**

When the gattc command queue full, the event comes

## BLUFI API

### Overview

BLUFI is a profile based GATT to config ESP32 WIFI to connect/disconnect AP or setup a softap and etc. Use should concern these things:

1. The event sent from profile. Then you need to do something as the event indicate.
2. Security reference. You can write your own Security functions such as symmetrical encryption/decryption and checksum functions. Even you can define the “Key Exchange/Negotiation” procedure.

### Application Example

Check [bluetooth](#) folder in ESP-IDF examples, which contains the following application:

- This is a BLUFI demo. This demo can set ESP32's wifi to softap/station/softap&station mode and config wifi connections - [bluetooth/blufi](#)

### API Reference

#### Header File

- [bt/bluedroid/api/include/esp\\_blufi\\_api.h](#)

## Functions

`esp_err_t esp_blufi_register_callbacks (esp_blufi_callbacks_t *callbacks)`

This function is called to receive blufi callback event.

**Return** ESP\_OK - success, other - failed

### Parameters

- `callbacks`: callback functions

`esp_err_t esp_blufi_profile_init (void)`

This function is called to initialize blufi\_profile.

**Return** ESP\_OK - success, other - failed

`esp_err_t esp_blufi_profile_deinit (void)`

This function is called to de-initialize blufi\_profile.

**Return** ESP\_OK - success, other - failed

`esp_err_t esp_blufi_send_wifi_conn_report (wifi_mode_t opmode, esp_blufi_sta_conn_state_t sta_conn_state, uint8_t softap_conn_num, esp_blufi_extra_info_t *extra_info)`

This function is called to send wifi connection report.

**Return** ESP\_OK - success, other - failed

### Parameters

- `opmode`: : wifi opmode
- `sta_conn_state`: : station is already in connection or not
- `softap_conn_num`: : softap connection number
- `extra_info`: : extra information, such as `sta_ssid`, `softap_ssid` and etc.

`uint16_t esp_blufi_get_version (void)`

Get BLUFI profile version.

**Return** Most 8bit significant is Great version, Least 8bit is Sub version

`esp_err_t esp_blufi_close (esp_gatts_if_t gatts_if, uint16_t conn_id)`

Close a connection a remote device.

### Return

- ESP\_OK : success
- other : failed

### Parameters

- `gatts_if`: GATT server access interface
- `conn_id`: connection ID to be closed.

## Unions

**union** `esp_blufi_cb_param_t`

*#include* <esp\_blufi\_api.h> BLUFI callback parameters union.

### Public Members

**struct** `esp_blufi_cb_param_t::blufi_init_finish_evt_param` **init\_finish**  
Blufi callback param of ESP\_BLUFI\_EVENT\_INIT\_FINISH

**struct** `esp_blufi_cb_param_t::blufi_deinit_finish_evt_param` **deinit\_finish**  
Blufi callback param of ESP\_BLUFI\_EVENT\_DEINIT\_FINISH

**struct** `esp_blufi_cb_param_t::blufi_set_wifi_mode_evt_param` **wifi\_mode**  
Blufi callback param of ESP\_BLUFI\_EVENT\_INIT\_FINISH

**struct** `esp_blufi_cb_param_t::blufi_connect_evt_param` **connect**  
Blufi callback param of ESP\_BLUFI\_EVENT\_CONNECT

**struct** `esp_blufi_cb_param_t::blufi_disconnect_evt_param` **disconnect**  
Blufi callback param of ESP\_BLUFI\_EVENT\_DISCONNECT

**struct** `esp_blufi_cb_param_t::blufi_recv_sta_bssid_evt_param` **sta\_bssid**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_STA\_BSSID

**struct** `esp_blufi_cb_param_t::blufi_recv_sta_ssid_evt_param` **sta\_ssid**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_STA\_SSID

**struct** `esp_blufi_cb_param_t::blufi_recv_sta_passwd_evt_param` **sta\_passwd**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_STA\_PASSWD

**struct** `esp_blufi_cb_param_t::blufi_recv_softap_ssid_evt_param` **softap\_ssid**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_SSID

**struct** `esp_blufi_cb_param_t::blufi_recv_softap_passwd_evt_param` **softap\_passwd**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_PASSWD

**struct** `esp_blufi_cb_param_t::blufi_recv_softap_max_conn_num_evt_param` **softap\_max\_conn\_num**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_MAX\_CONN\_NUM

**struct** `esp_blufi_cb_param_t::blufi_recv_softap_auth_mode_evt_param` **softap\_auth\_mode**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_AUTH\_MODE

**struct** `esp_blufi_cb_param_t::blufi_recv_softap_channel_evt_param` **softap\_channel**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_CHANNEL

**struct** `esp_blufi_cb_param_t::blufi_recv_username_evt_param` **username**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_USERNAME

**struct** `esp_blufi_cb_param_t::blufi_recv_ca_evt_param` **ca**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_CA\_CERT

**struct** `esp_blufi_cb_param_t::blufi_recv_client_cert_evt_param` **client\_cert**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_CLIENT\_CERT

**struct** `esp_blufi_cb_param_t::blufi_recv_server_cert_evt_param` **server\_cert**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_SERVER\_CERT

**struct** `esp_blufi_cb_param_t::blufi_recv_client_pkey_evt_param` **client\_pkey**  
Blufi callback param of ESP\_BLUFI\_EVENT\_RECV\_CLIENT\_PRIV\_KEY

```
struct esp_blufi_cb_param_t::blufi_rcv_server_pkey_evt_param server_pkey
    Blufi callback param of ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY
```

```
struct blufi_connect_evt_param
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_CONNECT.
```

### Public Members

```
esp_bd_addr_t remote_bda
    Blufi Remote bluetooth device address
```

```
uint8_t server_if
    server interface
```

```
uint16_t conn_id
    Connection id
```

```
struct blufi_deinit_finish_evt_param
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_DEINIT_FINISH.
```

### Public Members

```
esp_blufi_deinit_state_t state
    De-initial status
```

```
struct blufi_disconnect_evt_param
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_DISCONNECT.
```

### Public Members

```
esp_bd_addr_t remote_bda
    Blufi Remote bluetooth device address
```

```
struct blufi_init_finish_evt_param
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_INIT_FINISH.
```

### Public Members

```
esp_blufi_init_state_t state
    Initial status
```

```
struct blufi_rcv_ca_evt_param
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CA_CERT.
```

### Public Members

```
uint8_t *cert
    CA certificate point
```

```
int cert_len
    CA certificate length
```

```
struct blufi_rcv_client_cert_evt_param
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_CLIENT_CERT
```

### Public Members

`uint8_t *cert`  
Client certificate point

`int cert_len`  
Client certificate length

**struct blufi\_recv\_client\_pkey\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_CLIENT\_PRIV\_KEY

### Public Members

`uint8_t *pkey`  
Client Private Key point, if Client certificate not contain Key

`int pkey_len`  
Client Private key length

**struct blufi\_recv\_server\_cert\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SERVER\_CERT

### Public Members

`uint8_t *cert`  
Client certificate point

`int cert_len`  
Client certificate length

**struct blufi\_recv\_server\_pkey\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SERVER\_PRIV\_KEY

### Public Members

`uint8_t *pkey`  
Client Private Key point, if Client certificate not contain Key

`int pkey_len`  
Client Private key length

**struct blufi\_recv\_softap\_auth\_mode\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_AUTH\_MODE.

### Public Members

`wifi_auth_mode_t auth_mode`  
Authentication mode

**struct blufi\_recv\_softap\_channel\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_CHANNEL.

### Public Members

uint8\_t **channel**  
Authentication mode

**struct blufi\_rcv\_softap\_max\_conn\_num\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_MAX\_CONN\_NUM.

### Public Members

int **max\_conn\_num**  
SSID

**struct blufi\_rcv\_softap\_passwd\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_PASSWD.

### Public Members

uint8\_t \***passwd**  
Password

int **passwd\_len**  
Password Length

**struct blufi\_rcv\_softap\_ssid\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_SSID.

### Public Members

uint8\_t \***ssid**  
SSID

int **ssid\_len**  
SSID length

**struct blufi\_rcv\_sta\_bssid\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_STA\_BSSID.

### Public Members

uint8\_t **bssid**[6]  
BSSID

**struct blufi\_rcv\_sta\_passwd\_evt\_param**  
*#include <esp\_blufi\_api.h>* ESP\_BLUFI\_EVENT\_RECV\_STA\_PASSWD.

### Public Members

uint8\_t \***passwd**  
Password

int **passwd\_len**  
Password Length

```
struct blufi_rcv_sta_ssid_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_STA_SSID.
```

### Public Members

```
uint8_t *ssid  
    SSID  
  
int ssid_len  
    SSID length
```

```
struct blufi_rcv_username_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_RECV_USERNAME.
```

### Public Members

```
uint8_t *name  
    Username point  
  
int name_len  
    Username length
```

```
struct blufi_set_wifi_mode_evt_param  
    #include <esp_blufi_api.h> ESP_BLUFI_EVENT_SET_WIFI_MODE.
```

### Public Members

```
wifi_mode_t op_mode  
    Wifi operation mode
```

## Structures

```
struct esp_blufi_extra_info_t  
    BLUFI extra information structure.
```

### Public Members

```
uint8_t sta_bssid[6]  
    BSSID of station interface  
  
bool sta_bssid_set  
    is BSSID of station interface set  
  
uint8_t *sta_ssid  
    SSID of station interface  
  
int sta_ssid_len  
    length of SSID of station interface  
  
uint8_t *sta_passwd  
    password of station interface  
  
int sta_passwd_len  
    length of password of station interface
```

```

uint8_t *softap_ssid
    SSID of softap interface

int softap_ssid_len
    length of SSID of softap interface

uint8_t *softap_passwd
    password of station interface

int softap_passwd_len
    length of password of station interface

uint8_t softap_authmode
    authentication mode of softap interface

bool softap_authmode_set
    is authentication mode of softap interface set

uint8_t softap_max_conn_num
    max connection number of softap interface

bool softap_max_conn_num_set
    is max connection number of softap interface set

uint8_t softap_channel
    channel of softap interface

bool softap_channel_set
    is channel of softap interface set

struct esp_blufi_callbacks_t
    BLUFI callback functions type.

```

## Public Members

```

esp_blufi_event_cb_t event_cb
    BLUFI event callback

esp_blufi_negotiate_data_handler_t negotiate_data_handler
    BLUFI negotiate data function for negotiate share key

esp_blufi_encrypt_func_t encrypt_func
    BLUFI encrypt data function with share key generated by negotiate_data_handler

esp_blufi_decrypt_func_t decrypt_func
    BLUFI decrypt data function with share key generated by negotiate_data_handler

esp_blufi_checksum_func_t checksum_func
    BLUFI check sum function (FCS)

```

## Type Definitions

```

typedef void (*esp_blufi_event_cb_t) (esp_blufi_cb_event_t event, esp_blufi_cb_param_t *param)
    BLUFI event callback function type.

```

### Parameters

- event: : Event type

- param: : Point to callback parameter, currently is union type

```
typedef void (*esp_blufi_negotiate_data_handler_t) (uint8_t *data, int len, uint8_t **out-
                                                    put_data, int *output_len, bool
                                                    *need_free)
```

BLUFI negotiate data handler.

#### Parameters

- data: : data from phone
- len: : length of data from phone
- output\_data: : data want to send to phone
- output\_len: : length of data want to send to phone

```
typedef int (*esp_blufi_encrypt_func_t) (uint8_t iv8, uint8_t *crypt_data, int cyprt_len)
BLUFI encrypt the data after negotiate a share key.
```

**Return** Nonnegative number is encrypted length, if error, return negative number;

#### Parameters

- iv8: : initial vector(8bit), normally, blufi core will input packet sequence number
- crypt\_data: : plain text and encrypted data, the encrypt function must support autochthonous encrypt
- crypt\_len: : length of plain text

```
typedef int (*esp_blufi_decrypt_func_t) (uint8_t iv8, uint8_t *crypt_data, int crypt_len)
BLUFI decrypt the data after negotiate a share key.
```

**Return** Nonnegative number is decrypted length, if error, return negative number;

#### Parameters

- iv8: : initial vector(8bit), normally, blufi core will input packet sequence number
- crypt\_data: : encrypted data and plain text, the encrypt function must support autochthonous decrypt
- crypt\_len: : length of encrypted text

```
typedef uint16_t (*esp_blufi_checksum_func_t) (uint8_t iv8, uint8_t *data, int len)
BLUFI checksum.
```

#### Parameters

- iv8: : initial vector(8bit), normally, blufi core will input packet sequence number
- data: : data need to checksum
- len: : length of data

## Enumerations

```
enum esp_blufi_cb_event_t
```

*Values:*

```
ESP_BLUFI_EVENT_INIT_FINISH = 0
ESP_BLUFI_EVENT_DEINIT_FINISH
ESP_BLUFI_EVENT_SET_WIFI_OPMODE
ESP_BLUFI_EVENT_BLE_CONNECT
ESP_BLUFI_EVENT_BLE_DISCONNECT
ESP_BLUFI_EVENT_REQ_CONNECT_TO_AP
ESP_BLUFI_EVENT_REQ_DISCONNECT_FROM_AP
ESP_BLUFI_EVENT_GET_WIFI_STATUS
ESP_BLUFI_EVENT_DEAUTHENTICATE_STA
ESP_BLUFI_EVENT_RECV_STA_BSSID
ESP_BLUFI_EVENT_RECV_STA_SSID
ESP_BLUFI_EVENT_RECV_STA_PASSWD
ESP_BLUFI_EVENT_RECV_SOFTAP_SSID
ESP_BLUFI_EVENT_RECV_SOFTAP_PASSWD
ESP_BLUFI_EVENT_RECV_SOFTAP_MAX_CONN_NUM
ESP_BLUFI_EVENT_RECV_SOFTAP_AUTH_MODE
ESP_BLUFI_EVENT_RECV_SOFTAP_CHANNEL
ESP_BLUFI_EVENT_RECV_USERNAME
ESP_BLUFI_EVENT_RECV_CA_CERT
ESP_BLUFI_EVENT_RECV_CLIENT_CERT
ESP_BLUFI_EVENT_RECV_SERVER_CERT
ESP_BLUFI_EVENT_RECV_CLIENT_PRIV_KEY
ESP_BLUFI_EVENT_RECV_SERVER_PRIV_KEY
ESP_BLUFI_EVENT_RECV_SLAVE_DISCONNECT_BLE
```

```
enum esp_blufi_sta_conn_state_t
    BLUFI config status.
```

*Values:*

```
ESP_BLUFI_STA_CONN_SUCCESS = 0x00
ESP_BLUFI_STA_CONN_FAIL = 0x01
```

```
enum esp_blufi_init_state_t
    BLUFI init status.
```

*Values:*

```
ESP_BLUFI_INIT_OK = 0
ESP_BLUFI_INIT_FAILED = 0
```

```
enum esp_blufi_deinit_state_t
    BLUFI deinit status.
```

*Values:*

```
ESP_BLUFI_DEINIT_OK = 0
ESP_BLUFI_DEINIT_FAILED = 0
```

## 2.2.4 CLASSIC BT

### CLASSIC BLUETOOTH GAP API

#### Overview

Instructions

#### Application Example

Instructions

#### API Reference

##### Header File

- `bt/bluedroid/api/include/esp_gap_bt_api.h`

##### Functions

`esp_err_t esp_bt_gap_set_scan_mode(esp_bt_scan_mode_t mode)`

Set discoverability and connectability mode for legacy bluetooth. This function should be called after `esp_bluedroid_enable()` completes successfully.

##### Return

- `ESP_OK` : Succeed
- `ESP_ERR_INVALID_ARG`: if argument invalid
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

##### Parameters

- `mode`: : one of the enums of `bt_scan_mode_t`

##### Enumerations

`enum esp_bt_scan_mode_t`

Discoverability and Connectability mode.

*Values:*

```
ESP_BT_SCAN_MODE_NONE = 0
    Neither discoverable nor connectable
```

**ESP\_BT\_SCAN\_MODE\_CONNECTABLE**

Connectable but not discoverable

**ESP\_BT\_SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE**

both discoverable and connectable

## Bluetooth A2DP API

### Overview

Instructions

### Application Example

Check `bluetooth` folder in ESP-IDF examples, which contains the following application:

- This is a A2DP sink client demo. This demo can be discovered and connected by A2DP source device and receive the audio stream from remote device - `bluetooth/a2dp_sink`

### API Reference

#### Header File

- `bt/bluedroid/api/include/esp_a2dp_api.h`

#### Functions

`esp_err_t esp_a2d_register_callback` (*esp\_a2d\_cb\_t* callback)

Register application callback function to A2DP module. This function should be called only after `esp_bluedroid_enable()` completes successfully.

#### Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: if callback is a NULL function pointer

#### Parameters

- `callback`: A2DP sink event callback function

`esp_err_t esp_a2d_register_data_callback` (*esp\_a2d\_data\_cb\_t* callback)

Register A2DP sink data output function; For now the output is PCM data stream decoded from SBC format. This function should be called only after `esp_bluedroid_enable()` completes successfully, used only by A2DP sink. The callback is invoked in the context of A2DP sink task whose stack size is configurable through menu-config.

#### Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled

- ESP\_FAIL: if callback is a NULL function pointer

**Parameters**

- `callback`: A2DP data callback function

`esp_err_t esp_a2d_sink_init` (void)

Initialize the bluetooth A2DP sink module. This function should be called after `esp_bluedroid_enable()` completes successfully.

**Return**

- ESP\_OK: if the initialization request is sent successfully
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

`esp_err_t esp_a2d_sink_deinit` (void)

De-initialize for A2DP sink module. This function should be called only after `esp_bluedroid_enable()` completes successfully.

**Return**

- ESP\_OK: success
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

`esp_err_t esp_a2d_sink_connect` (*esp\_bd\_addr\_t remote\_bda*)

Connect the remote bluetooth device bluetooth, must after `esp_a2d_sink_init()`

**Return**

- ESP\_OK: connect request is sent to lower layer
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

**Parameters**

- `remote_bda`: remote bluetooth device address

`esp_err_t esp_a2d_sink_disconnect` (*esp\_bd\_addr\_t remote\_bda*)

Disconnect the remote bluetooth device.

**Return**

- ESP\_OK: disconnect request is sent to lower layer
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

**Parameters**

- `remote_bda`: remote bluetooth device address

## Unions

```
union esp_a2d_cb_param_t
    #include <esp_a2dp_api.h> A2DP state callback parameters.
```

### Public Members

```
struct esp_a2d_cb_param_t::a2d_conn_stat_param conn_stat
    A2DP connection status

struct esp_a2d_cb_param_t::a2d_audio_stat_param audio_stat
    audio stream playing state

struct esp_a2d_cb_param_t::a2d_audio_cfg_param audio_cfg
    media codec configuration infomation

struct a2d_audio_cfg_param
    #include <esp_a2dp_api.h> ESP_A2D_AUDIO_CFG_EVT.
```

### Public Members

```
esp_bd_addr_t remote_bda
    remote bluetooth device address

esp_a2d_mcc_t mcc
    A2DP media codec capability information

struct a2d_audio_stat_param
    #include <esp_a2dp_api.h> ESP_A2D_AUDIO_STATE_EVT.
```

### Public Members

```
esp_a2d_audio_state_t state
    one of the values from esp_a2d_audio_state_t

esp_bd_addr_t remote_bda
    remote bluetooth device address

struct a2d_conn_stat_param
    #include <esp_a2dp_api.h> ESP_A2D_CONNECTION_STATE_EVT.
```

### Public Members

```
esp_a2d_connection_state_t state
    one of values from esp_a2d_connection_state_t

esp_bd_addr_t remote_bda
    remote bluetooth device address

esp_a2d_disc_rsn_t disc_rsn
    reason of disconnection for “DISCONNECTED”
```

## Structures

**struct esp\_a2d\_mcc\_t**  
A2DP media codec capabilities union.

### Public Members

**esp\_a2d\_mct\_t type**  
A2DP media codec type

**union esp\_a2d\_mcc\_t::[anonymous] cie**  
A2DP codec information element

## Macros

**ESP\_A2D\_MCT\_SBC**  
Media codec types supported by A2DP.  
SBC

**ESP\_A2D\_MCT\_M12**  
MPEG-1, 2 Audio

**ESP\_A2D\_MCT\_M24**  
MPEG-2, 4 AAC

**ESP\_A2D\_MCT\_ATRAC**  
ATRAC family

**ESP\_A2D\_MCT\_NON\_A2DP**

**ESP\_A2D\_CIE\_LEN\_SBC**

**ESP\_A2D\_CIE\_LEN\_M12**

**ESP\_A2D\_CIE\_LEN\_M24**

**ESP\_A2D\_CIE\_LEN\_ATRAC**

## Type Definitions

**typedef uint8\_t esp\_a2d\_mct\_t**

**typedef void (\*esp\_a2d\_cb\_t) (esp\_a2d\_cb\_event\_t event, esp\_a2d\_cb\_param\_t \*param)**  
A2DP profile callback function type.

### Parameters

- event: : Event type
- param: : Pointer to callback parameter

**typedef void (\*esp\_a2d\_data\_cb\_t) (const uint8\_t \*buf, uint32\_t len)**  
A2DP profile data callback function.

### Parameters

- `buf`: : data received from A2DP source device and is PCM format decoder from SBC decoder; `buf` references to a static memory block and can be overwritten by upcoming data
- `len`: : size(in bytes) in `buf`

## Enumerations

### `enum esp_a2d_connection_state_t`

Bluetooth A2DP connection states.

*Values:*

**ESP\_A2D\_CONNECTION\_STATE\_DISCONNECTED** = 0  
connection released

**ESP\_A2D\_CONNECTION\_STATE\_CONNECTING**  
connecting remote device

**ESP\_A2D\_CONNECTION\_STATE\_CONNECTED**  
connection established

**ESP\_A2D\_CONNECTION\_STATE\_DISCONNECTING**  
disconnecting remote device

### `enum esp_a2d_disc_rsn_t`

Bluetooth A2DP disconnection reason.

*Values:*

**ESP\_A2D\_DISC\_RSN\_NORMAL** = 0  
Finished disconnection that is initiated by local or remote device

**ESP\_A2D\_DISC\_RSN\_ABNORMAL**  
Abnormal disconnection caused by signal loss

### `enum esp_a2d_audio_state_t`

Bluetooth A2DP datapath states.

*Values:*

**ESP\_A2D\_AUDIO\_STATE\_REMOTE\_SUSPEND** = 0  
audio stream datapath suspended by remote device

**ESP\_A2D\_AUDIO\_STATE\_STOPPED**  
audio stream datapath stopped

**ESP\_A2D\_AUDIO\_STATE\_STARTED**  
audio stream datapath started

### `enum esp_a2d_cb_event_t`

A2DP callback events.

*Values:*

**ESP\_A2D\_CONNECTION\_STATE\_EVT** = 0  
connection state changed event

**ESP\_A2D\_AUDIO\_STATE\_EVT** = 1  
audio stream transmission state changed event

**ESP\_A2D\_AUDIO\_CFG\_EVT** = 2  
audio codec is configured

## BT AVRCP APIs

### Overview

Bluetooth AVRCP reference APIs.

[Instructions](#)

### Application Example

[Instructions](#)

### API Reference

#### Header File

- `bt/bluedroid/api/include/esp_avrc_api.h`

#### Functions

`esp_err_t esp_avrc_ct_register_callback` (*esp\_avrc\_ct\_cb\_t* callback)

Register application callbacks to AVRCP module; for now only AVRCP Controller role is supported. This function should be called after `esp_bluedroid_enable()` completes successfully.

##### Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

##### Parameters

- `callback`: AVRCP controller callback function

`esp_err_t esp_avrc_ct_init` (void)

Initialize the bluetooth AVRCP controller module, This function should be called after `esp_bluedroid_enable()` completes successfully.

##### Return

- `ESP_OK`: success
- `ESP_INVALID_STATE`: if bluetooth stack is not yet enabled
- `ESP_FAIL`: others

`esp_err_t esp_avrc_ct_deinit` (void)

De-initialize AVRCP controller module. This function should be called after after `esp_bluedroid_enable()` completes successfully.

##### Return

- `ESP_OK`: success

- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

esp\_err\_t **esp\_avrc\_ct\_send\_set\_player\_value\_cmd** (uint8\_t *tl*, uint8\_t *attr\_id*, uint8\_t *value\_id*)

Send player application settings command to AVRCP target. This function should be called after ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT is received and AVRCP connection is established.

#### Return

- ESP\_OK: success
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

#### Parameters

- *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- *attr\_id*: : player application setting attribute IDs from one of esp\_avrc\_ps\_attr\_ids\_t
- *value\_id*: : attribute value defined for the specific player application setting attribute

esp\_err\_t **esp\_avrc\_ct\_send\_register\_notification\_cmd** (uint8\_t *tl*, uint8\_t *event\_id*, uint32\_t *event\_parameter*)

Send register notification command to AVRCP target, This function should be called after ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT is received and AVRCP connection is established.

#### Return

- ESP\_OK: success
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

#### Parameters

- *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- *event\_id*: : id of events, e.g. ESP\_AVRC\_RN\_PLAY\_STATUS\_CHANGE, ESP\_AVRC\_RN\_TRACK\_CHANGE, etc.
- *event\_parameter*: : special parameters, eg. playback interval for ESP\_AVRC\_RN\_PLAY\_POS\_CHANGED

esp\_err\_t **esp\_avrc\_ct\_send\_metadata\_cmd** (uint8\_t *tl*, uint8\_t *attr\_mask*)

Send metadata command to AVRCP target, This function should be called after ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT is received and AVRCP connection is established.

#### Return

- ESP\_OK: success
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

#### Parameters

- *tl*: : transaction label, 0 to 15, consecutive commands should use different values.
- *attr\_mask*: : mask of attributes, e.g. ESP\_AVRC\_MD\_ATTR\_ID\_TITLE | ESP\_AVRC\_MD\_ATTR\_ID\_ARTIST.

`esp_err_t esp_avrc_ct_send_passthrough_cmd (uint8_t tl, uint8_t key_code, uint8_t key_state)`  
 Send passthrough command to AVRCP target, This function should be called after ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT is received and AVRCP connection is established.

#### Return

- ESP\_OK: success
- ESP\_INVALID\_STATE: if bluetooth stack is not yet enabled
- ESP\_FAIL: others

#### Parameters

- `tl`: : transaction label, 0 to 15, consecutive commands should use different values.
- `key_code`: : passthrough command code, e.g. ESP\_AVRC\_PT\_CMD\_PLAY, ESP\_AVRC\_PT\_CMD\_STOP, etc.
- `key_state`: : passthrough command key state, ESP\_AVRC\_PT\_CMD\_STATE\_PRESSED or ESP\_AVRC\_PT\_CMD\_STATE\_RELEASED

## Unions

`union esp_avrc_ct_cb_param_t`  
*#include <esp\_avrc\_api.h>* AVRCP controller callback parameters.

#### Public Members

`struct esp_avrc_ct_cb_param_t::avrc_ct_conn_stat_param conn_stat`  
 AVRCP connection status

`struct esp_avrc_ct_cb_param_t::avrc_ct_psth_rsp_param psth_rsp`  
 passthrough command response

`struct esp_avrc_ct_cb_param_t::avrc_ct_meta_rsp_param meta_rsp`  
 metadata attributes response

`struct esp_avrc_ct_cb_param_t::avrc_ct_change_notify_param change_ntf`  
 notifications

`struct esp_avrc_ct_cb_param_t::avrc_ct_rmt_feats_param rmt_feats`  
 AVRCP features discovered from remote SDP server

`struct avrc_ct_change_notify_param`  
*#include <esp\_avrc\_api.h>* ESP\_AVRC\_CT\_CHANGE\_NOTIFY\_EVT.

#### Public Members

`uint8_t event_id`  
 id of AVRCP event notification

`uint32_t event_parameter`  
 event notification parameter

`struct avrc_ct_conn_stat_param`  
*#include <esp\_avrc\_api.h>* ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT.

### Public Members

bool **connected**  
whether AVRC connection is set up

*esp\_bd\_addr\_t* **remote\_bda**  
remote bluetooth device address

**struct avrc\_ct\_meta\_rsp\_param**  
*#include <esp\_avrc\_api.h>* ESP\_AVRC\_CT\_METADATA\_RSP\_EVT.

### Public Members

uint8\_t **attr\_id**  
id of metadata attribute

uint8\_t \***attr\_text**  
attribute itself

int **attr\_length**  
attribute character length

**struct avrc\_ct\_psth\_rsp\_param**  
*#include <esp\_avrc\_api.h>* ESP\_AVRC\_CT\_PASSTHROUGH\_RSP\_EVT.

### Public Members

uint8\_t **t1**  
transaction label, 0 to 15

uint8\_t **key\_code**  
passthrough command code

uint8\_t **key\_state**  
0 for PRESSED, 1 for RELEASED

**struct avrc\_ct\_rmt\_feats\_param**  
*#include <esp\_avrc\_api.h>* ESP\_AVRC\_CT\_REMOTE\_FEATURES\_EVT.

### Public Members

uint32\_t **feat\_mask**  
AVRC feature mask of remote device

*esp\_bd\_addr\_t* **remote\_bda**  
remote bluetooth device address

### Type Definitions

**typedef** void (\***esp\_avrc\_ct\_cb\_t**)(*esp\_avrc\_ct\_cb\_event\_t* event, *esp\_avrc\_ct\_cb\_param\_t* \*param)  
AVRCP controller callback function type.

### Parameters

- event: : Event type
- param: : Pointer to callback parameter union

## Enumerations

### enum esp\_avrc\_features\_t

AVRC feature bit mask.

*Values:*

- ESP\_AVRC\_FEAT\_RCTG** = 0x0001  
remote control target
- ESP\_AVRC\_FEAT\_RCCT** = 0x0002  
remote control controller
- ESP\_AVRC\_FEAT\_VENDOR** = 0x0008  
remote control vendor dependent commands
- ESP\_AVRC\_FEAT\_BROWSE** = 0x0010  
use browsing channel
- ESP\_AVRC\_FEAT\_META\_DATA** = 0x0040  
remote control metadata transfer command/response
- ESP\_AVRC\_FEAT\_ADV\_CTRL** = 0x0200  
remote control advanced control commmand/response

### enum esp\_avrc\_pt\_cmd\_t

AVRC passthrough command code.

*Values:*

- ESP\_AVRC\_PT\_CMD\_PLAY** = 0x44  
play
- ESP\_AVRC\_PT\_CMD\_STOP** = 0x45  
stop
- ESP\_AVRC\_PT\_CMD\_PAUSE** = 0x46  
pause
- ESP\_AVRC\_PT\_CMD\_FORWARD** = 0x4B  
forward
- ESP\_AVRC\_PT\_CMD\_BACKWARD** = 0x4C  
backward
- ESP\_AVRC\_PT\_CMD\_REWIND** = 0x48  
rewind
- ESP\_AVRC\_PT\_CMD\_FAST\_FORWARD** = 0x49  
fast forward

### enum esp\_avrc\_pt\_cmd\_state\_t

AVRC passthrough command state.

*Values:*

- ESP\_AVRC\_PT\_CMD\_STATE\_PRESSED** = 0  
key pressed

**ESP\_AVRC\_PT\_CMD\_STATE\_RELEASED = 1**  
key released

**enum esp\_avrc\_ct\_cb\_event\_t**  
AVRC Controller callback events.

*Values:*

**ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT = 0**  
connection state changed event

**ESP\_AVRC\_CT\_PASSTHROUGH\_RSP\_EVT = 1**  
passthrough response event

**ESP\_AVRC\_CT\_METADATA\_RSP\_EVT = 2**  
metadata response event

**ESP\_AVRC\_CT\_PLAY\_STATUS\_RSP\_EVT = 3**  
play status response event

**ESP\_AVRC\_CT\_CHANGE\_NOTIFY\_EVT = 4**  
notification event

**ESP\_AVRC\_CT\_REMOTE\_FEATURES\_EVT = 5**  
feature of remote device indication event

**enum esp\_avrc\_md\_attr\_mask\_t**  
AVRC metadata attribute mask.

*Values:*

**ESP\_AVRC\_MD\_ATTR\_TITLE = 0x1**  
title of the playing track

**ESP\_AVRC\_MD\_ATTR\_ARTIST = 0x2**  
track artist

**ESP\_AVRC\_MD\_ATTR\_ALBUM = 0x4**  
album name

**ESP\_AVRC\_MD\_ATTR\_TRACK\_NUM = 0x8**  
track position on the album

**ESP\_AVRC\_MD\_ATTR\_NUM\_TRACKS = 0x10**  
number of tracks on the album

**ESP\_AVRC\_MD\_ATTR\_GENRE = 0x20**  
track genre

**ESP\_AVRC\_MD\_ATTR\_PLAYING\_TIME = 0x40**  
total album playing time in milliseconds

**enum esp\_avrc\_rn\_event\_ids\_t**  
AVRC event notification ids.

*Values:*

**ESP\_AVRC\_RN\_PLAY\_STATUS\_CHANGE = 0x01**  
track status change, eg. from playing to paused

**ESP\_AVRC\_RN\_TRACK\_CHANGE = 0x02**  
new track is loaded

**ESP\_AVRC\_RN\_TRACK\_REACHED\_END = 0x03**  
current track reached end

**ESP\_AVRC\_RN\_TRACK\_REACHED\_START** = 0x04  
current track reached start position

**ESP\_AVRC\_RN\_PLAY\_POS\_CHANGED** = 0x05  
track playing position changed

**ESP\_AVRC\_RN\_BATTERY\_STATUS\_CHANGE** = 0x06  
battery status changed

**ESP\_AVRC\_RN\_SYSTEM\_STATUS\_CHANGE** = 0x07  
system status changed

**ESP\_AVRC\_RN\_APP\_SETTING\_CHANGE** = 0x08  
application settings changed

**ESP\_AVRC\_RN\_MAX\_EVT**

**enum esp\_avrc\_ps\_attr\_ids\_t**  
AVRC player setting ids.

*Values:*

**ESP\_AVRC\_PS\_EQUALIZER** = 0x01  
equalizer, on or off

**ESP\_AVRC\_PS\_REPEAT\_MODE** = 0x02  
repeat mode

**ESP\_AVRC\_PS\_SHUFFLE\_MODE** = 0x03  
shuffle mode

**ESP\_AVRC\_PS\_SCAN\_MODE** = 0x04  
scan mode on or off

**ESP\_AVRC\_PS\_MAX\_ATTR**

**enum esp\_avrc\_ps\_eq\_value\_ids\_t**  
AVRC equalizer modes.

*Values:*

**ESP\_AVRC\_PS\_EQUALIZER\_OFF** = 0x1  
equalizer OFF

**ESP\_AVRC\_PS\_EQUALIZER\_ON** = 0x2  
equalizer ON

**enum esp\_avrc\_ps\_rpt\_value\_ids\_t**  
AVRC repeat modes.

*Values:*

**ESP\_AVRC\_PS\_REPEAT\_OFF** = 0x1  
repeat mode off

**ESP\_AVRC\_PS\_REPEAT\_SINGLE** = 0x2  
single track repeat

**ESP\_AVRC\_PS\_REPEAT\_GROUP** = 0x3  
group repeat

**enum esp\_avrc\_ps\_shf\_value\_ids\_t**  
AVRC shuffle modes.

*Values:*

```

ESP_AVRC_PS_SHUFFLE_OFF = 0x1
ESP_AVRC_PS_SHUFFLE_ALL = 0x2
ESP_AVRC_PS_SHUFFLE_GROUP = 0x3

```

```
enum esp_avrc_ps_scn_value_ids_t
```

AVRC scan modes.

*Values:*

```

ESP_AVRC_PS_SCAN_OFF = 0x1
    scan off

ESP_AVRC_PS_SCAN_ALL = 0x2
    all tracks scan

ESP_AVRC_PS_SCAN_GROUP = 0x3
    group scan

```

Example code for this API section is provided in [bluetooth](#) directory of ESP-IDF examples.

## 2.3 Ethernet API

### 2.3.1 ETHERNET

#### Application Example

Ethernet example: [ethernet/ethernet](#).

#### PHY Interfaces

The configured PHY model(s) are set in software by configuring the `eth_config_t` structure for the given PHY.

Headers include a default configuration structure. These default configurations will need some members overridden or re-set before they can be used for a particular PHY hardware configuration. Consult the Ethernet example to see how this is done.

- [ethernet/include/eth\\_phy/phy.h](#) (common)
- [ethernet/include/eth\\_phy/phy\\_tlk110.h](#)
- [ethernet/include/eth\\_phy/phy\\_lan8720.h](#)

#### PHY Configuration Constants

```
const eth_config_t phy_tlk110_default_ethernet_config
```

Default TLK110 PHY configuration.

This configuration is not suitable for use as-is, it will need to be modified for your particular PHY hardware setup.

Consult the Ethernet example to see how this is done.

```
const eth_config_t phy_lan8720_default_ethernet_config
```

Default LAN8720 PHY configuration.

This configuration is not suitable for use as-is, it will need to be modified for your particular PHY hardware setup.

Consult the Ethernet example to see how this is done.

## API Reference - Ethernet

### Header File

- `ethernet/include/esp_eth.h`

### Functions

`esp_err_t esp_eth_init (eth_config_t *config)`

Init ethernet mac.

**Note** config can not be NULL, and phy chip must be suitable to phy init func.

#### Return

- ESP\_OK
- ESP\_FAIL

#### Parameters

- config: mac init data.

`esp_err_t esp_eth_init_internal (eth_config_t *config)`

Init Ethernet mac driver only.

For the most part, you need not call this function directly. It gets called from `esp_eth_init()`.

This function may be called, if you only need to initialize the Ethernet driver without having to use the network stack on top.

**Note** config can not be NULL, and phy chip must be suitable to phy init func.

#### Return

- ESP\_OK
- ESP\_FAIL

#### Parameters

- config: mac init data.

`esp_err_t esp_eth_tx (uint8_t *buf, uint16_t size)`

Send packet from tcp/ip to mac.

**Note** buf can not be NULL, size must be less than 1580

#### Return

- ESP\_OK
- ESP\_FAIL

#### Parameters

- `buf`: start address of packet data.
- `size`: size (byte) of packet data.

`esp_err_t esp_eth_enable` (void)  
Enable ethernet interface.

**Note** Shout be called after `esp_eth_init`

**Return**

- `ESP_OK`
- `ESP_FAIL`

`esp_err_t esp_eth_disable` (void)  
Disable ethernet interface.

**Note** Shout be called after `esp_eth_init`

**Return**

- `ESP_OK`
- `ESP_FAIL`

void `esp_eth_get_mac` (uint8\_t *mac*[6])  
Get mac addr.

Get mac of ethernet interface.

**Note** `mac` addr must be a valid unicast address

**Parameters**

- `mac`: start address of mac address.
- `mac`: store mac of the interface.

void `esp_eth_smi_write` (uint32\_t *reg\_num*, uint16\_t *value*)  
Read phy reg with smi interface.

**Note** phy base addr must be right.

**Parameters**

- `reg_num`: phy reg num.
- `value`: value which write to phy reg.

uint16\_t `esp_eth_smi_read` (uint32\_t *reg\_num*)  
Read phy reg with smi interface.

**Note** phy base addr must be right.

**Return** value what read from phy reg

**Parameters**

- `reg_num`: phy reg num.

`esp_err_t esp_eth_smi_wait_value` (`uint32_t reg_num`, `uint16_t value`, `uint16_t value_mask`, `int timeout_ms`)

Continuously read a PHY register over SMI interface, wait until the register has the desired value.

**Note** PHY base address must be right.

**Return** ESP\_OK if desired value matches, ESP\_ERR\_TIMEOUT if timed out.

**Parameters**

- `reg_num`: PHY register number
- `value`: Value to wait for (masked with `value_mask`)
- `value_mask`: Mask of bits to match in the register.
- `timeout_ms`: Timeout to wait for this value (milliseconds). 0 means never timeout.

**static** `esp_err_t esp_eth_smi_wait_set` (`uint32_t reg_num`, `uint16_t value_mask`, `int timeout_ms`)

Continuously read a PHY register over SMI interface, wait until the register has all bits in a mask set.

**Note** PHY base address must be right.

**Return** ESP\_OK if desired value matches, ESP\_ERR\_TIMEOUT if timed out.

**Parameters**

- `reg_num`: PHY register number
- `value_mask`: Value mask to wait for (all bits in this mask must be set)
- `timeout_ms`: Timeout to wait for this value (milliseconds). 0 means never timeout.

`void esp_eth_free_rx_buf` (`void *buf`)

Free emac rx buf.

**Note** buf can not be null, and it is tcpip input buf.

**Parameters**

- `buf`: start address of receive packet data.

`esp_err_t esp_eth_set_mac` (`const uint8_t mac[6]`)

Set mac of ethernet interface.

**Note** user can call this function after `emac_init`, and the new mac address will be enabled after `emac_enable`.

**Return**

- ESP\_OK: succeed
- ESP\_ERR\_INVALID\_MAC: invalid mac address

**Parameters**

- `mac`: the Mac address.

## Structures

**struct eth\_config\_t**

ethernet configuration

## Public Members

*eth\_phy\_base\_t* **phy\_addr**  
phy base addr (0~31)

*eth\_mode\_t* **mac\_mode**  
mac mode only support RMII now

*eth\_clock\_mode\_t* **clock\_mode**  
external/internal clock mode selecton

*eth\_tcpip\_input\_func* **tcpip\_input**  
tcpip input func

*eth\_phy\_func* **phy\_init**  
phy init func

*eth\_phy\_check\_link\_func* **phy\_check\_link**  
phy check link func

*eth\_phy\_check\_init\_func* **phy\_check\_init**  
phy check init func

*eth\_phy\_get\_speed\_mode\_func* **phy\_get\_speed\_mode**  
phy check init func

*eth\_phy\_get\_duplex\_mode\_func* **phy\_get\_duplex\_mode**  
phy check init func

*eth\_gpio\_config\_func* **gpio\_config**  
gpio config func

bool **flow\_ctrl\_enable**  
flag of flow ctrl enable

*eth\_phy\_get\_partner\_pause\_enable\_func* **phy\_get\_partner\_pause\_enable**  
get partner pause enable

*eth\_phy\_power\_enable\_func* **phy\_power\_enable**  
enable or disable phy power

## Type Definitions

```
typedef bool (*eth_phy_check_link_func) (void)
typedef void (*eth_phy_check_init_func) (void)
typedef eth_speed_mode_t (*eth_phy_get_speed_mode_func) (void)
typedef eth_duplex_mode_t (*eth_phy_get_duplex_mode_func) (void)
typedef void (*eth_phy_func) (void)
typedef esp_err_t (*eth_tcpip_input_func) (void *buffer, uint16_t len, void *eb)
typedef void (*eth_gpio_config_func) (void)
typedef bool (*eth_phy_get_partner_pause_enable_func) (void)
typedef void (*eth_phy_power_enable_func) (bool enable)
```

## Enumerations

`enum eth_mode_t`

*Values:*

`ETH_MODE_RMII = 0`

`ETH_MODE_MII`

`enum eth_clock_mode_t`

*Values:*

`ETH_CLOCK_GPIO0_IN = 0`

`ETH_CLOCK_GPIO0_OUT = 1`

`ETH_CLOCK_GPIO16_OUT = 2`

`ETH_CLOCK_GPIO17_OUT = 3`

`enum eth_speed_mode_t`

*Values:*

`ETH_SPEED_MODE_10M = 0`

`ETH_SPEED_MODE_100M`

`enum eth_duplex_mode_t`

*Values:*

`ETH_MODE_HALFDUPLEX = 0`

`ETH_MODE_FULLLDUPLEX`

`enum eth_phy_base_t`

*Values:*

`PHY0 = 0`

`PHY1`

`PHY2`

`PHY3`

`PHY4`

`PHY5`

`PHY6`

`PHY7`

`PHY8`

`PHY9`

`PHY10`

`PHY11`

`PHY12`

`PHY13`

`PHY14`

`PHY15`

PHY16  
PHY17  
PHY18  
PHY19  
PHY20  
PHY21  
PHY22  
PHY23  
PHY24  
PHY25  
PHY26  
PHY27  
PHY28  
PHY29  
PHY30  
PHY31

## API Reference - PHY Common

### Header File

- [ethernet/include/eth\\_phy/phy.h](#)

### Functions

void **phy\_rmii\_configure\_data\_interface\_pins** (void)  
Common PHY-management functions.

These are not enough to drive any particular Ethernet PHY, but they provide a common configuration structure and management functions. Configure fixed pins for RMII data interface.

This configures GPIOs 0, 19, 22, 25, 26, 27 for use with RMII data interface. These pins cannot be changed, and must be wired to ethernet functions.

This is not sufficient to fully configure the Ethernet PHY, MDIO configuration interface pins (such as SMI MDC, MDO, MDI) must also be configured correctly in the GPIO matrix.

void **phy\_rmii\_smi\_configure\_pins** (uint8\_t *mdc\_gpio*, uint8\_t *mdio\_gpio*)  
Configure variable pins for SMI (MDIO) ethernet functions.

Calling this function along with `mii_configure_default_pins()` will fully configure the GPIOs for the ethernet PHY.

void **phy\_mii\_enable\_flow\_ctrl** (void)  
Enable flow control in standard PHY MII register.

bool **phy\_mii\_check\_link\_status** (void)

bool `phy_mii_get_partner_pause_enable` (void)

## API Reference - PHY TLK110

### Header File

- `ethernet/include/eth_phy/phy_tlk110.h`

### Functions

void `phy_tlk110_dump_registers` ()  
Dump all TLK110 PHY SMI configuration registers.

**Note** These registers are dumped at ‘debug’ level, so output may not be visible depending on default log levels.

void `phy_tlk110_check_phy_init` (void)  
Default TLK110 `phy_check_init` function.

*eth\_speed\_mode\_t* `phy_tlk110_get_speed_mode` (void)  
Default TLK110 `phy_get_speed_mode` function.

*eth\_duplex\_mode\_t* `phy_tlk110_get_duplex_mode` (void)  
Default TLK110 `phy_get_duplex_mode` function.

void `phy_tlk110_power_enable` (bool)  
Default TLK110 `phy_power_enable` function.

Consult the ethernet example to see how this is done.

**Note** This function may need to be replaced with a custom function if the PHY has a GPIO to enable power or start a clock.

void `phy_tlk110_init` (void)  
Default TLK110 `phy_init` function.

## API Reference - PHY LAN8720

### Header File

- `ethernet/include/eth_phy/phy_lan8720.h`

### Functions

void `phy_lan8720_dump_registers` ()  
Dump all LAN8720 PHY SMI configuration registers.

**Note** These registers are dumped at ‘debug’ level, so output may not be visible depending on default log levels.

void `phy_lan8720_check_phy_init` (void)  
Default LAN8720 `phy_check_init` function.

*eth\_speed\_mode\_t* `phy_lan8720_get_speed_mode` (void)  
Default LAN8720 `phy_get_speed_mode` function.

`eth_duplex_mode_t phy_lan8720_get_duplex_mode` (void)

Default LAN8720 `phy_get_duplex_mode` function.

void `phy_lan8720_power_enable` (bool)

Default LAN8720 `phy_power_enable` function.

Consult the ethernet example to see how this is done.

**Note** This function may need to be replaced with a custom function if the PHY has a GPIO to enable power or start a clock.

void `phy_lan8720_init` (void)

Default LAN8720 `phy_init` function.

Example code for this API section is provided in `ethernet` directory of ESP-IDF examples.

## 2.4 Peripherals API

### 2.4.1 Analog to Digital Converter

#### Overview

ESP32 integrates two 12-bit SAR (Successive Approximation Register) ADCs (Analog to Digital Converters) and supports measurements on 18 channels (analog enabled pins). Some of these pins can be used to build a programmable gain amplifier which is used for the measurement of small analog signals.

The ADC driver API supports ADC1 (9 channels, attached to GPIOs 32 - 39), and ADC2 (10 channels, attached to GPIOs 0, 2, 4, 12 - 15 and 25 - 27). However, there're some restrictions for the application to use ADC2:

1. The application can use ADC2 only when Wi-Fi driver is not started, since the ADC is also used by the Wi-Fi driver, which has higher priority.
2. Some of the ADC2 pins are used as strapping pins (GPIO 0, 2, 15), so they cannot be used freely. For examples, for official Develop Kits:
  - ESP32 Core Board V2 / ESP32 DevKitC: GPIO 0 cannot be used due to external auto program circuits.
  - ESP-WROVER-KIT V3: GPIO 0, 2, 4 and 15 cannot be used due to external connections for different purposes.

#### Configuration and Reading ADC

The ADC should be configured before reading is taken.

- For ADC1, configure desired precision and attenuation by calling functions `adc1_config_width()` and `adc1_config_channel_atten()`.
- For ADC2, configure the attenuation by `adc2_config_channel_atten()`. The reading width of ADC2 is configured every time you take the reading.

Attenuation configuration is done per channel, see `adc1_channel_t` and `adc2_channel_t`, set as a parameter of above functions.

Then it is possible to read ADC conversion result with `adc1_get_raw()` and `adc2_get_raw()`. Reading width of ADC2 should be set as a parameter of `adc2_get_raw()` instead of in the configuration functions.

**Note:** Since the ADC2 is shared with the WIFI module, which has higher priority, reading operation of `adc2_get_raw()` will fail between `esp_wifi_start()` and `esp_wifi_stop()`. Use the return code to see whether the reading is successful.

---

It is also possible to read the internal hall effect sensor via ADC1 by calling dedicated function `hall_sensor_read()`. Note that even the hall sensor is internal to ESP32, reading from it uses channels 0 and 3 of ADC1 (GPIO 36 and 39). Do not connect anything else to these pins and do not change their configuration. Otherwise it may affect the measurement of low value signal from the sensor.

This API provides convenient way to configure ADC1 for reading from *ULP*. To do so, call function `adc1_ulp_enable()` and then set precision and attenuation as discussed above.

There is another specific function `adc2_vref_to_gpio()` used to route internal reference voltage to a GPIO pin. It comes handy to calibrate ADC reading and this is discussed in section *Minimizing Noise*.

## Application Examples

Reading voltage on ADC1 channel 0 (GPIO 36):

```
#include <driver/adc.h>

...

adc1_config_width(ADC_WIDTH_BIT_12);
adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_DB_0);
int val = adc1_get_raw(ADC1_CHANNEL_0);
```

The input voltage in above example is from 0 to 1.1V (0 dB attenuation). The input range can be extended by setting higher attenuation, see `adc_atten_t`. An example using the ADC driver including calibration (discussed below) is available in esp-idf: [peripherals/adc](#)

Reading voltage on ADC2 channel 7 (GPIO 27):

```
#include <driver/adc.h>

...

int read_raw;
adc2_config_channel_atten( ADC2_CHANNEL_7, ADC_ATTEN_0db );

esp_err_t r = adc2_get_raw( ADC2_CHANNEL_7, ADC_WIDTH_12Bit, &read_raw);
if ( r == ESP_OK ) {
    printf("%d\n", read_raw );
} else if ( r == ESP_ERR_TIMEOUT ) {
    printf("ADC2 used by Wi-Fi.\n");
}
```

The reading may fail due to collision with Wi-Fi, should check it. An example using the ADC2 driver to read the output of DAC is available in esp-idf: [peripherals/adc2](#)

Reading the internal hall effect sensor:

```
#include <driver/adc.h>

...
```

(continues on next page)

(continued from previous page)

```
adc1_config_width(ADC_WIDTH_BIT_12);  
int val = hall_sensor_read();
```

The value read in both these examples is 12 bits wide (range 0-4095).

## Minimizing Noise

The ESP32 ADC can be sensitive to noise leading to large discrepancies in ADC readings. To minimize noise, users may connect a 0.1uF capacitor to the ADC input pad in use. Multisampling may also be used to further mitigate the effects of noise.

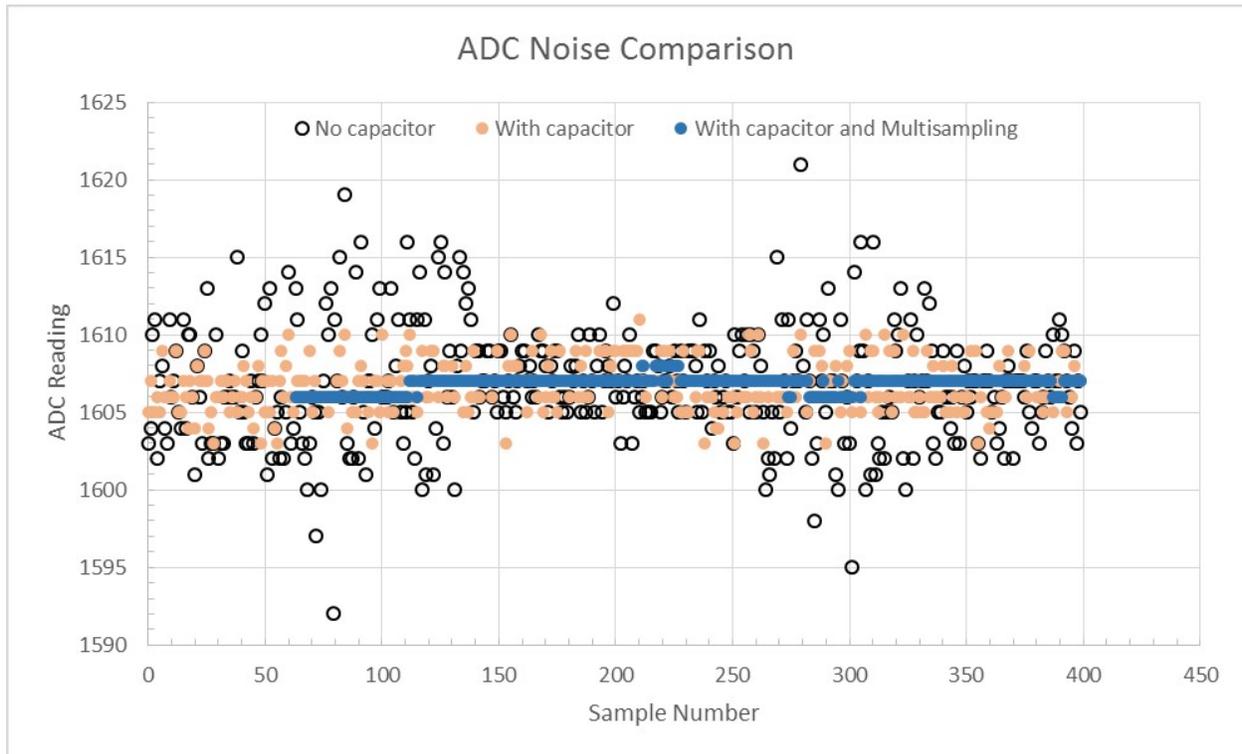


Fig. 1: Graph illustrating noise mitigation using capacitor and multisampling of 64 samples.

## ADC Calibration

The `esp_adc_cal/include/esp_adc_cal.h` API provides functions to correct for differences in measured voltages caused by variation of ADC reference voltages ( $V_{ref}$ ) between chips. Per design the ADC reference voltage is 1100mV, however the true reference voltage can range from 1000mV to 1200mV amongst different ESP32s.

Correcting ADC readings using this API involves characterizing one of the ADCs at a given attenuation to obtain a characteristics curve (ADC-Voltage curve) that takes into account the difference in ADC reference voltage. The characteristics curve is in the form of  $y = \text{coeff\_a} * x + \text{coeff\_b}$  and is used to convert ADC readings to voltages in mV. Calculation of the characteristics curve is based on calibration values which can be stored in eFuse or provided by the user.

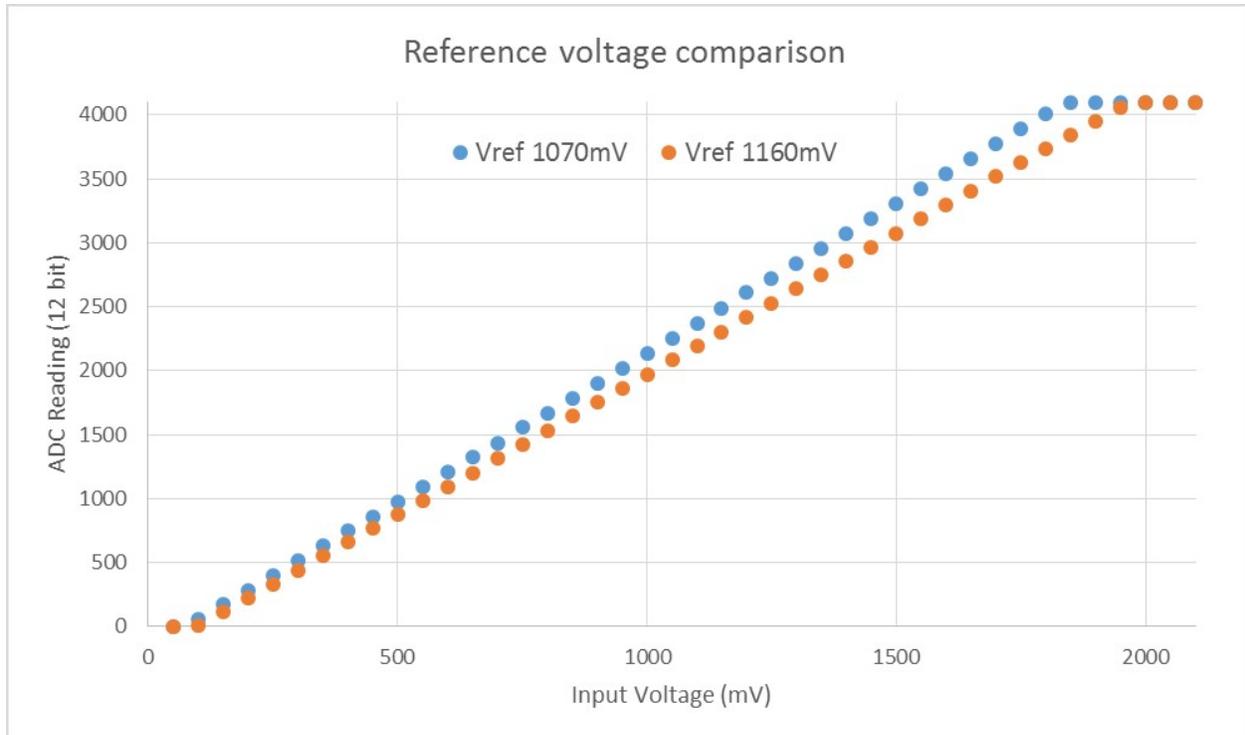


Fig. 2: Graph illustrating effect of differing reference voltages on the ADC voltage curve.

## Calibration Values

Calibration values are used to generate characteristic curves that account for the unique ADC reference voltage of a particular ESP32. There are currently three sources of calibration values. The availability of these calibration values will depend on the type and production date of the ESP32 chip/module.

**Two Point** values represent each of the ADCs' readings at 150mV and 850mV. These values are measured and burned into eFuse BLOCK3 during factory calibration.

**eFuse Vref** represents the true ADC reference voltage. This value is measured and burned into eFuse BLOCK0 during factory calibration.

**Default Vref** is an estimate of the ADC reference voltage provided by the user as a parameter during characterization. If Two Point or eFuse Vref values are unavailable, **Default Vref** will be used.

## Application Example

For a full example see esp-idf: [peripherals/adc](#)

Characterizing an ADC at a particular attenuation:

```
#include "driver/adc.h"
#include "esp_adc_cal.h"

...

//Characterize ADC at particular atten
esp_adc_cal_characteristics_t *adc_chars = calloc(1, sizeof(esp_adc_cal_
↪characteristics_t));
```

(continues on next page)

(continued from previous page)

```

esp_adc_cal_value_t val_type = esp_adc_cal_characterize(unit, atten, ADC_WIDTH_
↪BIT_12, DEFAULT_VREF, adc_chars);
//Check type of calibration value used to characterize ADC
if (val_type == ESP_ADC_CAL_VAL_EFUSE_VREF) {
    printf("eFuse Vref");
} else if (val_type == ESP_ADC_CAL_VAL_EFUSE_TP) {
    printf("Two Point");
} else {
    printf("Default");
}

```

Reading an ADC then converting the reading to a voltage:

```

#include "driver/adc.h"
#include "esp_adc_cal.h"

...

uint32_t reading = adc1_get_raw(ADC1_CHANNEL_5);
uint32_t voltage = esp_adc_cal_raw_to_voltage(reading, adc_chars);

```

Routing ADC reference voltage to GPIO, so it can be manually measured (for **Default Vref**):

```

#include "driver/adc.h"

...

esp_err_t status = adc2_vref_to_gpio(GPIO_NUM_25);
if (status == ESP_OK) {
    printf("v_ref routed to GPIO\n");
} else {
    printf("failed to route v_ref\n");
}

```

## GPIO Lookup Macros

There are macros available to specify the GPIO number of a ADC channel, or vice versa. e.g.

1. ADC1\_CHANNEL\_0\_GPIO\_NUM is the GPIO number of ADC1 channel 0 (36);
2. ADC1\_GPIO32\_CHANNEL is the ADC1 channel number of GPIO 32 (ADC1 channel 4).

## API Reference

This reference covers three components:

- *ADC driver*
- *ADC Calibration*
- *GPIO Lookup Macros*

## ADC driver

## Header File

- `driver/include/driver/adc.h`

## Functions

`esp_err_t adc1_pad_get_io_num(adc1_channel_t channel, gpio_num_t *gpio_num)`  
Get the gpio number of a specific ADC1 channel.

### Return

- `ESP_OK` if success
- `ESP_ERR_INVALID_ARG` if channel not valid

### Parameters

- `channel`: Channel to get the gpio number
- `gpio_num`: output buffer to hold the gpio number

`esp_err_t adc1_config_width(adc_bits_width_t width_bit)`  
Configure ADC1 capture width, meanwhile enable output invert for ADC1. The configuration is for all channels of ADC1.

### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` Parameter error

### Parameters

- `width_bit`: Bit capture width for ADC1

`esp_err_t adc_set_data_width(adc_unit_t adc_unit, adc_bits_width_t width_bit)`  
Configure ADC capture width.

### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` Parameter error

### Parameters

- `adc_unit`: ADC unit index
- `width_bit`: Bit capture width for ADC unit.

`esp_err_t adc1_config_channel_atten(adc1_channel_t channel, adc_atten_t atten)`  
Configure the ADC1 channel, including setting attenuation.

The default ADC full-scale voltage is 1.1V. To read higher voltages (up to the pin maximum voltage, usually 3.3V) requires setting >0dB signal attenuation for that ADC channel.

**Note** This function also configures the input GPIO pin mux to connect it to the ADC1 channel. It must be called before calling `adc1_get_raw()` for this channel.

When `VDD_A` is 3.3V:

- 0dB attenuation (ADC\_ATTEN\_DB\_0) gives full-scale voltage 1.1V
- 2.5dB attenuation (ADC\_ATTEN\_DB\_2\_5) gives full-scale voltage 1.5V
- 6dB attenuation (ADC\_ATTEN\_DB\_6) gives full-scale voltage 2.2V
- 11dB attenuation (ADC\_ATTEN\_DB\_11) gives full-scale voltage 3.9V (see note below)

**Note** The full-scale voltage is the voltage corresponding to a maximum reading (depending on ADC1 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

**Note** At 11dB attenuation the maximum voltage is limited by VDD\_A, not the full scale voltage.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `channel`: ADC1 channel to configure
- `atten`: Attenuation level

int **adc1\_get\_raw** (*adc1\_channel\_t channel*)

Take an ADC1 reading on a single channel.

**Note** Call `adc1_config_width()` before the first time this function is called.

**Note** For a given channel, `adc1_config_channel_atten(channel)` must be called before the first time this function is called.

#### Return

- -1: Parameter error
- Other: ADC1 channel reading.

#### Parameters

- `channel`: ADC1 channel to read

void **adc\_power\_on** ()

Enable ADC power.

void **adc\_power\_off** ()

Power off SAR ADC This function will force power down for ADC.

esp\_err\_t **adc\_gpio\_init** (*adc\_unit\_t adc\_unit, adc\_channel\_t channel*)

Initialize ADC pad.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `adc_unit`: ADC unit index
- `channel`: ADC channel index

esp\_err\_t **adc\_set\_data\_inv** (*adc\_unit\_t* *adc\_unit*, bool *inv\_en*)  
Set ADC data invert.

**Return**

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *adc\_unit*: ADC unit index
- *inv\_en*: whether enable data invert

esp\_err\_t **adc\_set\_clk\_div** (uint8\_t *clk\_div*)  
Set ADC source clock.

**Return**

- ESP\_OK success

**Parameters**

- *clk\_div*: ADC clock divider, ADC clock is divided from APB clock

esp\_err\_t **adc\_set\_i2s\_data\_source** (*adc\_i2s\_source\_t* *src*)  
Set I2S data source.

**Return**

- ESP\_OK success

**Parameters**

- *src*: I2S DMA data source, I2S DMA can get data from digital signals or from ADC.

esp\_err\_t **adc\_i2s\_mode\_init** (*adc\_unit\_t* *adc\_unit*, *adc\_channel\_t* *channel*)  
Initialize I2S ADC mode.

**Return**

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *adc\_unit*: ADC unit index
- *channel*: ADC channel index

void **adc1\_ulp\_enable** ()  
Configure ADC1 to be usable by the ULP.

This function reconfigures ADC1 to be controlled by the ULP. Effect of this function can be reverted using `adc1_get_raw` function.

Note that `adc1_config_channel_atten`, `adc1_config_width` functions need to be called to configure ADC1 channels, before ADC1 is used by the ULP.

int **hall\_sensor\_read** ()  
Read Hall Sensor.

**Note** The Hall Sensor uses channels 0 and 3 of ADC1. Do not configure these channels for use as ADC channels.

**Note** The ADC1 module must be enabled by calling `adc1_config_width()` before calling `hall_sensor_read()`. ADC1 should be configured for 12 bit readings, as the hall sensor readings are low values and do not cover the full range of the ADC.

**Return** The hall sensor reading.

`esp_err_t adc2_pad_get_io_num(adc2_channel_t channel, gpio_num_t *gpio_num)`  
Get the gpio number of a specific ADC2 channel.

**Return**

- ESP\_OK if success
- ESP\_ERR\_INVALID\_ARG if channel not valid

**Parameters**

- `channel`: Channel to get the gpio number
- `gpio_num`: output buffer to hold the gpio number

`esp_err_t adc2_config_channel_atten(adc2_channel_t channel, adc_atten_t atten)`  
Configure the ADC2 channel, including setting attenuation.

The default ADC full-scale voltage is 1.1V. To read higher voltages (up to the pin maximum voltage, usually 3.3V) requires setting >0dB signal attenuation for that ADC channel.

**Note** This function also configures the input GPIO pin mux to connect it to the ADC2 channel. It must be called before calling `adc2_get_raw()` for this channel.

When VDD\_A is 3.3V:

- 0dB attenuation (`ADC_ATTEN_0db`) gives full-scale voltage 1.1V
- 2.5dB attenuation (`ADC_ATTEN_2_5db`) gives full-scale voltage 1.5V
- 6dB attenuation (`ADC_ATTEN_6db`) gives full-scale voltage 2.2V
- 11dB attenuation (`ADC_ATTEN_11db`) gives full-scale voltage 3.9V (see note below)

**Note** The full-scale voltage is the voltage corresponding to a maximum reading (depending on ADC2 configured bit width, this value is: 4095 for 12-bits, 2047 for 11-bits, 1023 for 10-bits, 511 for 9 bits.)

**Note** At 11dB attenuation the maximum voltage is limited by VDD\_A, not the full scale voltage.

**Return**

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `channel`: ADC2 channel to configure
- `atten`: Attenuation level

`esp_err_t adc2_get_raw(adc2_channel_t channel, adc_bits_width_t width_bit, int *raw_out)`  
Take an ADC2 reading on a single channel.

**Note** For a given channel, `adc2_config_channel_atten()` must be called before the first time this function is called. If Wi-Fi is started via `esp_wifi_start()`, this function will always fail with `ESP_ERR_TIMEOUT`.

#### Return

- `ESP_OK` if success
- `ESP_ERR_TIMEOUT` the WIFI is started, using the ADC2

#### Parameters

- `channel`: ADC2 channel to read
- `width_bit`: Bit capture width for ADC2
- `raw_out`: the variable to hold the output data.

`esp_err_t adc2_vref_to_gpio(gpio_num_t gpio)`

Output ADC2 reference voltage to gpio 25 or 26 or 27.

This function utilizes the testing mux exclusive to ADC 2 to route the reference voltage one of ADC2's channels. Supported gpios are gpios 25, 26, and 27. This reference voltage can be manually read from the pin and used in the `esp_adc_cal` component.

#### Return

- `ESP_OK`: `v_ref` successfully routed to selected gpio
- `ESP_ERR_INVALID_ARG`: Unsupported gpio

#### Parameters

- `gpio`: GPIO number (gpios 25,26,27 supported)

## Macros

`ADC_ATTEN_0db`

`ADC_ATTEN_2_5db`

`ADC_ATTEN_6db`

`ADC_ATTEN_11db`

`ADC_WIDTH_9Bit`

`ADC_WIDTH_10Bit`

`ADC_WIDTH_11Bit`

`ADC_WIDTH_12Bit`

## Enumerations

`enum adc_atten_t`

*Values:*

`ADC_ATTEN_DB_0 = 0`

The input voltage of ADC will be reduced to about 1/1

**ADC\_ATTEN\_DB\_2\_5 = 1**

The input voltage of ADC will be reduced to about 1/1.34

**ADC\_ATTEN\_DB\_6 = 2**

The input voltage of ADC will be reduced to about 1/2

**ADC\_ATTEN\_DB\_11 = 3**

The input voltage of ADC will be reduced to about 1/3.6

**ADC\_ATTEN\_MAX**

**enum adc\_bits\_width\_t**

*Values:*

**ADC\_WIDTH\_BIT\_9 = 0**

ADC capture width is 9Bit

**ADC\_WIDTH\_BIT\_10 = 1**

ADC capture width is 10Bit

**ADC\_WIDTH\_BIT\_11 = 2**

ADC capture width is 11Bit

**ADC\_WIDTH\_BIT\_12 = 3**

ADC capture width is 12Bit

**ADC\_WIDTH\_MAX**

**enum adc1\_channel\_t**

*Values:*

**ADC1\_CHANNEL\_0 = 0**

ADC1 channel 0 is GPIO36

**ADC1\_CHANNEL\_1**

ADC1 channel 1 is GPIO37

**ADC1\_CHANNEL\_2**

ADC1 channel 2 is GPIO38

**ADC1\_CHANNEL\_3**

ADC1 channel 3 is GPIO39

**ADC1\_CHANNEL\_4**

ADC1 channel 4 is GPIO32

**ADC1\_CHANNEL\_5**

ADC1 channel 5 is GPIO33

**ADC1\_CHANNEL\_6**

ADC1 channel 6 is GPIO34

**ADC1\_CHANNEL\_7**

ADC1 channel 7 is GPIO35

**ADC1\_CHANNEL\_MAX**

**enum adc2\_channel\_t**

*Values:*

**ADC2\_CHANNEL\_0 = 0**

ADC2 channel 0 is GPIO4

**ADC2\_CHANNEL\_1**

ADC2 channel 1 is GPIO0

**ADC2\_CHANNEL\_2**  
ADC2 channel 2 is GPIO2

**ADC2\_CHANNEL\_3**  
ADC2 channel 3 is GPIO15

**ADC2\_CHANNEL\_4**  
ADC2 channel 4 is GPIO13

**ADC2\_CHANNEL\_5**  
ADC2 channel 5 is GPIO12

**ADC2\_CHANNEL\_6**  
ADC2 channel 6 is GPIO14

**ADC2\_CHANNEL\_7**  
ADC2 channel 7 is GPIO27

**ADC2\_CHANNEL\_8**  
ADC2 channel 8 is GPIO25

**ADC2\_CHANNEL\_9**  
ADC2 channel 9 is GPIO26

**ADC2\_CHANNEL\_MAX**

**enum adc\_channel\_t**

*Values:*

**ADC\_CHANNEL\_0** = 0  
ADC channel

**ADC\_CHANNEL\_1**  
ADC channel

**ADC\_CHANNEL\_2**  
ADC channel

**ADC\_CHANNEL\_3**  
ADC channel

**ADC\_CHANNEL\_4**  
ADC channel

**ADC\_CHANNEL\_5**  
ADC channel

**ADC\_CHANNEL\_6**  
ADC channel

**ADC\_CHANNEL\_7**  
ADC channel

**ADC\_CHANNEL\_8**  
ADC channel

**ADC\_CHANNEL\_9**  
ADC channel

**ADC\_CHANNEL\_MAX**

**enum adc\_unit\_t**

*Values:*

```

ADC_UNIT_1 = 1
    SAR ADC 1

ADC_UNIT_2 = 2
    SAR ADC 2, not supported yet

ADC_UNIT_BOTH = 3
    SAR ADC 1 and 2, not supported yet

ADC_UNIT_ALTER = 7
    SAR ADC 1 and 2 alternative mode, not supported yet

ADC_UNIT_MAX

```

```
enum adc_i2s_encode_t
```

*Values:*

```

ADC_ENCODE_12BIT
    ADC to I2S data format, [15:12]-channel [11:0]-12 bits ADC data

ADC_ENCODE_11BIT
    ADC to I2S data format, [15]-1 [14:11]-channel [10:0]-11 bits ADC data

ADC_ENCODE_MAX

```

```
enum adc_i2s_source_t
```

*Values:*

```

ADC_I2S_DATA_SRC_IO_SIG = 0
    I2S data from GPIO matrix signal

ADC_I2S_DATA_SRC_ADC = 1
    I2S data from ADC

ADC_I2S_DATA_SRC_MAX

```

## ADC Calibration

### Header File

- `esp_adc_cal/include/esp_adc_cal.h`

### Functions

```
esp_err_t esp_adc_cal_check_efuse(esp_adc_cal_value_t value_type)
```

Checks if ADC calibration values are burned into eFuse.

This function checks if ADC reference voltage or Two Point values have been burned to the eFuse of the current ESP32

#### Return

- `ESP_OK`: The calibration mode is supported in eFuse
- `ESP_ERR_NOT_SUPPORTED`: Error, eFuse values are not burned
- `ESP_ERR_INVALID_ARG`: Error, invalid argument (`ESP_ADC_CAL_VAL_DEFAULT_VREF`)

#### Parameters

- `value_type`: Type of calibration value (ESP\_ADC\_CAL\_VAL\_EFUSE\_VREF or ESP\_ADC\_CAL\_VAL\_EFUSE\_TP)

`esp_adc_cal_value_t` **esp\_adc\_cal\_characterize** (`adc_unit_t` `adc_num`, `adc_atten_t` `atten`, `adc_bits_width_t` `bit_width`, `uint32_t` `default_vref`, `esp_adc_cal_characteristics_t` `*chars`)

Characterize an ADC at a particular attenuation.

This function will characterize the ADC at a particular attenuation and generate the ADC-Voltage curve in the form of  $[y = \text{coeff\_a} * x + \text{coeff\_b}]$ . Characterization can be based on Two Point values, eFuse Vref, or default Vref and the calibration values will be prioritized in that order.

**Note** Two Point values and eFuse Vref can be enabled/disabled using menuconfig.

**Return**

- ESP\_ADC\_CAL\_VAL\_EFUSE\_VREF: eFuse Vref used for characterization
- ESP\_ADC\_CAL\_VAL\_EFUSE\_TP: Two Point value used for characterization (only in Linear Mode)
- ESP\_ADC\_CAL\_VAL\_DEFAULT\_VREF: Default Vref used for characterization

**Parameters**

- `adc_num`: ADC to characterize (ADC\_UNIT\_1 or ADC\_UNIT\_2)
- `atten`: Attenuation to characterize
- `bit_width`: Bit width configuration of ADC
- `default_vref`: Default ADC reference voltage in mV (used if eFuse values is not available)
- `chars`: Pointer to empty structure used to store ADC characteristics

`uint32_t` **esp\_adc\_cal\_raw\_to\_voltage** (`uint32_t` `adc_reading`, `const` `esp_adc_cal_characteristics_t` `*chars`)

Convert an ADC reading to voltage in mV.

This function converts an ADC reading to a voltage in mV based on the ADC's characteristics.

**Note** Characteristics structure must be initialized before this function is called (call `esp_adc_cal_characterize()`)

**Return** Voltage in mV

**Parameters**

- `adc_reading`: ADC reading
- `chars`: Pointer to initialized structure containing ADC characteristics

`esp_err_t` **esp\_adc\_cal\_get\_voltage** (`adc_channel_t` `channel`, `const` `esp_adc_cal_characteristics_t` `*chars`, `uint32_t` `*voltage`)

Reads an ADC and converts the reading to a voltage in mV.

This function reads an ADC then converts the raw reading to a voltage in mV based on the characteristics provided. The ADC that is read is also determined by the characteristics.

**Note** The Characteristics structure must be initialized before this function is called (call `esp_adc_cal_characterize()`)

**Return**

- ESP\_OK: ADC read and converted to mV

- `ESP_ERR_TIMEOUT`: Error, timed out attempting to read ADC
- `ESP_ERR_INVALID_ARG`: Error due to invalid arguments

#### Parameters

- `channel`: ADC Channel to read
- `chars`: Pointer to initialized ADC characteristics structure
- `voltage`: Pointer to store converted voltage

#### Structures

##### `struct esp_adc_cal_characteristics_t`

Structure storing characteristics of an ADC.

**Note** Call `esp_adc_cal_characterize()` to initialize the structure

#### Public Members

`adc_unit_t` **adc\_num**

ADC number

`adc_atten_t` **atten**

ADC attenuation

`adc_bits_width_t` **bit\_width**

ADC bit width

`uint32_t` **coeff\_a**

Gradient of ADC-Voltage curve

`uint32_t` **coeff\_b**

Offset of ADC-Voltage curve

`uint32_t` **vref**

Vref used by lookup table

**const** `uint32_t` \***low\_curve**

Pointer to low Vref curve of lookup table (NULL if unused)

**const** `uint32_t` \***high\_curve**

Pointer to high Vref curve of lookup table (NULL if unused)

#### Enumerations

##### `enum esp_adc_cal_value_t`

Type of calibration value used in characterization.

*Values:*

**ESP\_ADC\_CAL\_VAL\_EFUSE\_VREF** = 0

Characterization based on reference voltage stored in eFuse

**ESP\_ADC\_CAL\_VAL\_EFUSE\_TP** = 1

Characterization based on Two Point values stored in eFuse

`ESP_ADC_CAL_VAL_DEFAULT_VREF = 2`  
Characterization based on default reference voltage

## GPIO Lookup Macros

### Header File

- `soc/esp32/include/soc/adc_channel.h`

### Macros

`ADC1_GPIO36_CHANNEL`  
`ADC1_CHANNEL_0_GPIO_NUM`  
`ADC1_GPIO37_CHANNEL`  
`ADC1_CHANNEL_1_GPIO_NUM`  
`ADC1_GPIO38_CHANNEL`  
`ADC1_CHANNEL_2_GPIO_NUM`  
`ADC1_GPIO39_CHANNEL`  
`ADC1_CHANNEL_3_GPIO_NUM`  
`ADC1_GPIO32_CHANNEL`  
`ADC1_CHANNEL_4_GPIO_NUM`  
`ADC1_GPIO33_CHANNEL`  
`ADC1_CHANNEL_5_GPIO_NUM`  
`ADC1_GPIO34_CHANNEL`  
`ADC1_CHANNEL_6_GPIO_NUM`  
`ADC1_GPIO35_CHANNEL`  
`ADC1_CHANNEL_7_GPIO_NUM`  
`ADC2_GPIO4_CHANNEL`  
`ADC2_CHANNEL_0_GPIO_NUM`  
`ADC2_GPIO0_CHANNEL`  
`ADC2_CHANNEL_1_GPIO_NUM`  
`ADC2_GPIO2_CHANNEL`  
`ADC2_CHANNEL_2_GPIO_NUM`  
`ADC2_GPIO15_CHANNEL`  
`ADC2_CHANNEL_3_GPIO_NUM`  
`ADC2_GPIO13_CHANNEL`  
`ADC2_CHANNEL_4_GPIO_NUM`  
`ADC2_GPIO12_CHANNEL`

`ADC2_CHANNEL_5_GPIO_NUM``ADC2_GPIO14_CHANNEL``ADC2_CHANNEL_6_GPIO_NUM``ADC2_GPIO27_CHANNEL``ADC2_CHANNEL_7_GPIO_NUM``ADC2_GPIO25_CHANNEL``ADC2_CHANNEL_8_GPIO_NUM``ADC2_GPIO26_CHANNEL``ADC2_CHANNEL_9_GPIO_NUM`

## 2.4.2 Digital To Analog Converter

### Overview

ESP32 has two 8-bit DAC (digital to analog converter) channels, connected to GPIO25 (Channel 1) and GPIO26 (Channel 2).

The DAC driver allows these channels to be set to arbitrary voltages.

The DAC channels can also be driven with DMA-style written sample data, via the *I2S driver* when using the “built-in DAC mode”.

For other analog output options, see the *Sigma-delta Modulation module* and the *LED Control module*. Both these modules produce high frequency PWM output, which can be hardware low-pass filtered in order to generate a lower frequency analog output.

### Application Example

Setting DAC channel 1 (GPIO 25) voltage to approx 0.78 of VDD\_A voltage ( $VDD * 200 / 255$ ). For VDD\_A 3.3V, this is 2.59V:

```
#include <driver/dac.h>
...
dac_output_enable(DAC_CHANNEL_1);
dac_output_voltage(DAC_CHANNEL_1, 200);
```

### API Reference

#### Header File

- `driver/include/driver/dac.h`

#### Functions

`esp_err_t dac_pad_get_io_num(dac_channel_t channel, gpio_num_t *gpio_num)`

Get the gpio number of a specific DAC channel.

#### Return

- ESP\_OK if success
- ESP\_ERR\_INVALID\_ARG if channel not valid

#### Parameters

- channel: Channel to get the gpio number
- gpio\_num: output buffer to hold the gpio number

esp\_err\_t **dac\_output\_voltage** (*dac\_channel\_t channel*, uint8\_t *dac\_value*)  
Set DAC output voltage.

DAC output is 8-bit. Maximum (255) corresponds to VDD.

**Note** Need to configure DAC pad before calling this function. DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- channel: DAC channel
- dac\_value: DAC output value

esp\_err\_t **dac\_output\_enable** (*dac\_channel\_t channel*)  
DAC pad output enable.

**Note** DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26 I2S left channel will be mapped to DAC channel 2 I2S right channel will be mapped to DAC channel 1

#### Parameters

- channel: DAC channel

esp\_err\_t **dac\_output\_disable** (*dac\_channel\_t channel*)  
DAC pad output disable.

**Note** DAC channel 1 is attached to GPIO25, DAC channel 2 is attached to GPIO26

#### Parameters

- channel: DAC channel

esp\_err\_t **dac\_i2s\_enable** ()  
Enable DAC output data from I2S.

esp\_err\_t **dac\_i2s\_disable** ()  
Disable DAC output data from I2S.

## Enumerations

**enum dac\_channel\_t**

*Values:*

**DAC\_CHANNEL\_1 = 1**  
DAC channel 1 is GPIO25

**DAC\_CHANNEL\_2**  
DAC channel 2 is GPIO26

**DAC\_CHANNEL\_MAX**

## GPIO Lookup Macros

Some useful macros can be used to specified the GPIO number of a DAC channel, or vice versa. e.g.

1. `DAC_CHANNEL_1_GPIO_NUM` is the GPIO number of channel 1 (25);
2. `DAC_GPIO26_CHANNEL` is the channel number of GPIO 26 (channel 2).

## Header File

- `soc/esp32/include/soc/dac_channel.h`

## Macros

**DAC\_GPIO25\_CHANNEL**

**DAC\_CHANNEL\_1\_GPIO\_NUM**

**DAC\_GPIO26\_CHANNEL**

**DAC\_CHANNEL\_2\_GPIO\_NUM**

## 2.4.3 GPIO & RTC GPIO

### Overview

The ESP32 chip features 40 physical GPIO pads. Some GPIO pads cannot be used or do not have the corresponding pin on the chip package(refer to technical reference manual). Each pad can be used as a general purpose I/O or can be connected to an internal peripheral signal.

- Note that GPIO6-11 are usually used for SPI flash.
- GPIO34-39 can only be set as input mode and do not have software pullup or pulldown functions.

There is also separate “RTC GPIO” support, which functions when GPIOs are routed to the “RTC” low-power and analog subsystem. These pin functions can be used when in deep sleep, when the *Ultra Low Power co-processor* is running, or when analog functions such as ADC/DAC/etc are in use.

### Application Example

GPIO output and input interrupt example: [peripherals/gpio](#).

## API Reference - Normal GPIO

### Header File

- `driver/include/driver/gpio.h`

### Functions

`esp_err_t gpio_config (const gpio_config_t *pGPIOConfig)`  
GPIO common configuration.

Configure GPIO's Mode,pull-up,PullDown,IntrType

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- pGPIOConfig: Pointer to GPIO configure struct

`esp_err_t gpio_set_intr_type (gpio_num_t gpio_num, gpio_int_type_t intr_type)`  
GPIO set interrupt trigger type.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- gpio\_num: GPIO number. If you want to set the trigger type of e.g. of GPIO16, gpio\_num should be GPIO\_NUM\_16 (16);
- intr\_type: Interrupt type, select from gpio\_int\_type\_t

`esp_err_t gpio_intr_enable (gpio_num_t gpio_num)`  
Enable GPIO module interrupt signal.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- gpio\_num: GPIO number. If you want to enable an interrupt on e.g. GPIO16, gpio\_num should be GPIO\_NUM\_16 (16);

`esp_err_t gpio_intr_disable (gpio_num_t gpio_num)`  
Disable GPIO module interrupt signal.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `gpio_num`: GPIO number. If you want to disable the interrupt of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

`esp_err_t gpio_set_level` (*gpio\_num\_t gpio\_num, uint32\_t level*)  
GPIO set output level.

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` GPIO number error

**Parameters**

- `gpio_num`: GPIO number. If you want to set the output level of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `level`: Output level. 0: low ; 1: high

`int gpio_get_level` (*gpio\_num\_t gpio\_num*)  
GPIO get input level.

**Return**

- 0 the GPIO input level is 0
- 1 the GPIO input level is 1

**Parameters**

- `gpio_num`: GPIO number. If you want to get the logic level of e.g. pin GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);

`esp_err_t gpio_set_direction` (*gpio\_num\_t gpio\_num, gpio\_mode\_t mode*)  
GPIO set direction.

Configure GPIO direction, such as `output_only`, `input_only`, `output_and_input`

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` GPIO error

**Parameters**

- `gpio_num`: Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `mode`: GPIO direction

`esp_err_t gpio_set_pull_mode` (*gpio\_num\_t gpio\_num, gpio\_pull\_mode\_t pull*)  
Configure GPIO pull-up/pull-down resistors.

Only pins that support both input & output have integrated pull-up and pull-down resistors. Input-only GPIOs 34-39 do not.

**Return**

- `ESP_OK` Success

- ESP\_ERR\_INVALID\_ARG : Parameter error

#### Parameters

- `gpio_num`: GPIO number. If you want to set pull up or down mode for e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
- `pull`: GPIO pull up/down mode.

`esp_err_t gpio_wakeup_enable (gpio_num_t gpio_num, gpio_int_type_t intr_type)`  
Enable GPIO wake-up function.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number.
- `intr_type`: GPIO wake-up type. Only `GPIO_INTR_LOW_LEVEL` or `GPIO_INTR_HIGH_LEVEL` can be used.

`esp_err_t gpio_wakeup_disable (gpio_num_t gpio_num)`  
Disable GPIO wake-up function.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number

`esp_err_t gpio_isr_register (void (*fn)) void *`  
, void \*arg, int intr\_alloc\_flags, `gpio_isr_handle_t *handle` Register GPIO interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

This ISR function is called whenever any GPIO interrupt occurs. See the alternative `gpio_install_isr_service()` and `gpio_isr_handler_add()` API in order to have the driver support per-GPIO ISRs.

To disable or remove the ISR, pass the returned handle to the *interrupt allocation functions*.

#### Parameters

- `fn`: Interrupt handler function.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `arg`: Parameter for handler function
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

#### Return

- ESP\_OK Success ;
- ESP\_ERR\_INVALID\_ARG GPIO error

esp\_err\_t **gpio\_pullup\_en**(*gpio\_num\_t* *gpio\_num*)  
Enable pull-up on GPIO.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *gpio\_num*: GPIO number

esp\_err\_t **gpio\_pullup\_dis**(*gpio\_num\_t* *gpio\_num*)  
Disable pull-up on GPIO.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *gpio\_num*: GPIO number

esp\_err\_t **gpiopulldown\_en**(*gpio\_num\_t* *gpio\_num*)  
Enable pull-down on GPIO.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *gpio\_num*: GPIO number

esp\_err\_t **gpiopulldown\_dis**(*gpio\_num\_t* *gpio\_num*)  
Disable pull-down on GPIO.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *gpio\_num*: GPIO number

esp\_err\_t **gpio\_install\_isr\_service**(int *intr\_alloc\_flags*)  
Install the driver's GPIO ISR handler service, which allows per-pin GPIO interrupt handlers.

This function is incompatible with `gpio_isr_register()` - if that function is used, a single global ISR is registered for all GPIO interrupts. If this function is used, the ISR service provides a global GPIO ISR and individual pin handlers are registered via the `gpio_isr_handler_add()` function.

**Return**

- ESP\_OK Success

- ESP\_FAIL Operation fail
- ESP\_ERR\_NO\_MEM No memory to install this service

#### Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP\_INTR\_FLAG\_\* values. See `esp_intr_alloc.h` for more info.

void **gpio\_uninstall\_isr\_service** ()

Uninstall the driver's GPIO ISR service, freeing related resources.

esp\_err\_t **gpio\_isr\_handler\_add** (*gpio\_num\_t* gpio\_num, *gpio\_isr\_t* isr\_handler, void \*args)

Add ISR handler for the corresponding GPIO pin.

Call this function after using `gpio_install_isr_service()` to install the driver's GPIO ISR handler service.

The pin ISR handlers no longer need to be declared with `IRAM_ATTR`, unless you pass the `ESP_INTR_FLAG_IRAM` flag when allocating the ISR in `gpio_install_isr_service()`.

This ISR handler will be called from an ISR. So there is a stack size limit (configurable as "ISR stack size" in menuconfig). This limit is smaller compared to a global GPIO interrupt handler due to the additional level of indirection.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_STATE Wrong state, the ISR service has not been initialized.
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number
- `isr_handler`: ISR handler function for the corresponding GPIO number.
- `args`: parameter for ISR handler.

esp\_err\_t **gpio\_isr\_handler\_remove** (*gpio\_num\_t* gpio\_num)

Remove ISR handler for the corresponding GPIO pin.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_STATE Wrong state, the ISR service has not been initialized.
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number

esp\_err\_t **gpio\_set\_drive\_capability** (*gpio\_num\_t* gpio\_num, *gpio\_drive\_cap\_t* strength)

Set GPIO pad drive capability.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Drive capability of the pad

`esp_err_t gpio_get_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t *strength)`  
Get GPIO pad drive capability.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Pointer to accept drive capability of the pad

## Structures

**struct gpio\_config\_t**

Configuration parameters of GPIO pad for `gpio_config` function.

#### Public Members

`uint64_t pin_bit_mask`

GPIO pin: set with bit mask, each bit maps to a GPIO

`gpio_mode_t mode`

GPIO mode: set input/output mode

`gpio_pullup_t pull_up_en`

GPIO pull-up

`gpio_pulldown_t pull_down_en`

GPIO pull-down

`gpio_int_type_t intr_type`

GPIO interrupt type

## Macros

**GPIO\_SEL\_0**

Pin 0 selected

**GPIO\_SEL\_1**

Pin 1 selected

**GPIO\_SEL\_2**

Pin 2 selected

**Note** There are more macros like that up to pin 39, excluding pins 20, 24 and 28..31. They are not shown here to reduce redundant information.

**GPIO\_IS\_VALID\_GPIO** (`gpio_num`)

Check whether it is a valid GPIO number

`GPIO_IS_VALID_OUTPUT_GPIO` (gpio\_num)

Check whether it can be a valid GPIO number of output mode

## Type Definitions

```
typedef void (*gpio_isr_t) (void *)
```

```
typedef intr_handle_t gpio_isr_handle_t
```

## Enumerations

```
enum gpio_num_t
```

*Values:*

```
GPIO_NUM_0 = 0
```

GPIO0, input and output

```
GPIO_NUM_1 = 1
```

GPIO1, input and output

```
GPIO_NUM_2 = 2
```

GPIO2, input and output

**Note** There are more enumerations like that up to GPIO39, excluding GPIO20, GPIO24 and GPIO28..31. They are not shown here to reduce redundant information.

**Note** GPIO34..39 are input mode only.

```
enum gpio_int_type_t
```

*Values:*

```
GPIO_INTR_DISABLE = 0
```

Disable GPIO interrupt

```
GPIO_INTR_POSEDGE = 1
```

GPIO interrupt type : rising edge

```
GPIO_INTR_NEGEDGE = 2
```

GPIO interrupt type : falling edge

```
GPIO_INTR_ANYEDGE = 3
```

GPIO interrupt type : both rising and falling edge

```
GPIO_INTR_LOW_LEVEL = 4
```

GPIO interrupt type : input low level trigger

```
GPIO_INTR_HIGH_LEVEL = 5
```

GPIO interrupt type : input high level trigger

```
GPIO_INTR_MAX
```

```
enum gpio_mode_t
```

*Values:*

```
GPIO_MODE_DISABLE = GPIO_MODE_DEF_DISABLE
```

GPIO mode : disable input and output

```
GPIO_MODE_INPUT = GPIO_MODE_DEF_INPUT
```

GPIO mode : input only

**GPIO\_MODE\_OUTPUT** = GPIO\_MODE\_DEF\_OUTPUT

GPIO mode : output only mode

**GPIO\_MODE\_OUTPUT\_OD** = ((GPIO\_MODE\_DEF\_OUTPUT)|(GPIO\_MODE\_DEF\_OD))

GPIO mode : output only with open-drain mode

**GPIO\_MODE\_INPUT\_OUTPUT\_OD** = ((GPIO\_MODE\_DEF\_INPUT)|(GPIO\_MODE\_DEF\_OUTPUT)|(GPIO\_MODE\_DEF\_OD))

GPIO mode : output and input with open-drain mode

**GPIO\_MODE\_INPUT\_OUTPUT** = ((GPIO\_MODE\_DEF\_INPUT)|(GPIO\_MODE\_DEF\_OUTPUT))

GPIO mode : output and input mode

**enum gpio\_pullup\_t**

*Values:*

**GPIO\_PULLUP\_DISABLE** = 0x0

Disable GPIO pull-up resistor

**GPIO\_PULLUP\_ENABLE** = 0x1

Enable GPIO pull-up resistor

**enum gpio\_pulldown\_t**

*Values:*

**GPIO\_PULLDOWN\_DISABLE** = 0x0

Disable GPIO pull-down resistor

**GPIO\_PULLDOWN\_ENABLE** = 0x1

Enable GPIO pull-down resistor

**enum gpio\_pull\_mode\_t**

*Values:*

**GPIO\_PULLUP\_ONLY**

Pad pull up

**GPIO\_PULLDOWN\_ONLY**

Pad pull down

**GPIO\_PULLUP\_PULLDOWN**

Pad pull up + pull down

**GPIO\_FLOATING**

Pad floating

**enum gpio\_drive\_cap\_t**

*Values:*

**GPIO\_DRIVE\_CAP\_0** = 0

Pad drive capability: weak

**GPIO\_DRIVE\_CAP\_1** = 1

Pad drive capability: stronger

**GPIO\_DRIVE\_CAP\_2** = 2

Pad drive capability: default value

**GPIO\_DRIVE\_CAP\_DEFAULT** = 2

Pad drive capability: default value

**GPIO\_DRIVE\_CAP\_3** = 3

Pad drive capability: strongest

**GPIO\_DRIVE\_CAP\_MAX**

## API Reference - RTC GPIO

### Header File

- `driver/include/driver/rtc_io.h`

### Functions

**static** bool `rtc_gpio_is_valid_gpio` (*gpio\_num\_t* gpio\_num)

Determine if the specified GPIO is a valid RTC GPIO.

**Return** true if GPIO is valid for RTC GPIO use. false otherwise.

#### Parameters

- `gpio_num`: GPIO number

esp\_err\_t `rtc_gpio_init` (*gpio\_num\_t* gpio\_num)

Init a GPIO as RTC GPIO.

This function must be called when initializing a pad for an analog function.

#### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` GPIO is not an RTC IO

#### Parameters

- `gpio_num`: GPIO number (e.g. `GPIO_NUM_12`)

esp\_err\_t `rtc_gpio_deinit` (*gpio\_num\_t* gpio\_num)

Init a GPIO as digital GPIO.

#### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` GPIO is not an RTC IO

#### Parameters

- `gpio_num`: GPIO number (e.g. `GPIO_NUM_12`)

uint32\_t `rtc_gpio_get_level` (*gpio\_num\_t* gpio\_num)

Get the RTC IO input level.

#### Return

- 1 High level
- 0 Low level
- `ESP_ERR_INVALID_ARG` GPIO is not an RTC IO

#### Parameters

- `gpio_num`: GPIO number (e.g. `GPIO_NUM_12`)

`esp_err_t rtc_gpio_set_level` (*gpio\_num\_t* *gpio\_num*, *uint32\_t* *level*)  
Set the RTC IO output level.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- *gpio\_num*: GPIO number (e.g. GPIO\_NUM\_12)
- *level*: output level

`esp_err_t rtc_gpio_set_direction` (*gpio\_num\_t* *gpio\_num*, *rtc\_gpio\_mode\_t* *mode*)  
RTC GPIO set direction.

Configure RTC GPIO direction, such as output only, input only, output and input.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- *gpio\_num*: GPIO number (e.g. GPIO\_NUM\_12)
- *mode*: GPIO direction

`esp_err_t rtc_gpio_pullup_en` (*gpio\_num\_t* *gpio\_num*)  
RTC GPIO pullup enable.

This function only works for RTC IOs. In general, call `gpio_pullup_en`, which will work both for normal GPIOs and RTC IOs.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- *gpio\_num*: GPIO number (e.g. GPIO\_NUM\_12)

`esp_err_t rtc_gpio_pulldown_en` (*gpio\_num\_t* *gpio\_num*)  
RTC GPIO pulldown enable.

This function only works for RTC IOs. In general, call `gpio_pulldown_en`, which will work both for normal GPIOs and RTC IOs.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- *gpio\_num*: GPIO number (e.g. GPIO\_NUM\_12)

`esp_err_t rtc_gpio_pullup_dis (gpio_num_t gpio_num)`

RTC GPIO pullup disable.

This function only works for RTC IOs. In general, call `gpio_pullup_dis`, which will work both for normal GPIOs and RTC IOs.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- `gpio_num`: GPIO number (e.g. GPIO\_NUM\_12)

`esp_err_t rtc_gpiopulldown_dis (gpio_num_t gpio_num)`

RTC GPIO pulldown disable.

This function only works for RTC IOs. In general, call `gpiopulldown_dis`, which will work both for normal GPIOs and RTC IOs.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- `gpio_num`: GPIO number (e.g. GPIO\_NUM\_12)

`esp_err_t rtc_gpio_hold_en (gpio_num_t gpio_num)`

Enable hold function on an RTC IO pad.

Enabling HOLD function will cause the pad to latch current values of input enable, output enable, output value, function, drive strength values. This function is useful when going into light or deep sleep mode to prevent the pin configuration from changing.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- `gpio_num`: GPIO number (e.g. GPIO\_NUM\_12)

`esp_err_t rtc_gpio_hold_dis (gpio_num_t gpio_num)`

Disable hold function on an RTC IO pad.

Disabling hold function will allow the pad receive the values of input enable, output enable, output value, function, drive strength from RTC\_IO peripheral.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG GPIO is not an RTC IO

**Parameters**

- `gpio_num`: GPIO number (e.g. GPIO\_NUM\_12)

```
void rtc_gpio_force_hold_dis_all ()
```

Disable 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_set_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t strength)
```

Set RTC GPIO pad drive capability.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Drive capability of the pad

```
esp_err_t rtc_gpio_get_drive_capability (gpio_num_t gpio_num, gpio_drive_cap_t *strength)
```

Get RTC GPIO pad drive capability.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `gpio_num`: GPIO number, only support output GPIOs
- `strength`: Pointer to accept drive capability of the pad

## Structures

```
struct rtc_gpio_desc_t
```

Pin function information for a single GPIO pad’s RTC functions.

This is an internal function of the driver, and is not usually useful for external use.

#### Public Members

```
uint32_t reg
```

Register of RTC pad, or 0 if not an RTC GPIO

```
uint32_t mux
```

Bit mask for selecting digital pad or RTC pad

```
uint32_t func
```

Shift of pad function (FUN\_SEL) field

```
uint32_t ie
```

Mask of input enable

```
uint32_t pullup
```

Mask of pullup enable

`uint32_t pulldown`  
Mask of pulldown enable

`uint32_t slpsel`  
If `slpsel` bit is set, `slpie` will be used as pad input enabled signal in sleep mode

`uint32_t slpie`  
Mask of input enable in sleep mode

`uint32_t hold`  
Mask of hold enable

`uint32_t hold_force`  
Mask of `hold_force` bit for RTC IO in `RTC_CNTL_HOLD_FORCE_REG`

`uint32_t drv_v`  
Mask of drive capability

`uint32_t drv_s`  
Offset of drive capability

`int rtc_num`  
RTC IO number, or -1 if not an RTC GPIO

## Macros

`RTC_GPIO_IS_VALID_GPIO` (`gpio_num`)

## Enumerations

`enum rtc_gpio_mode_t`

*Values:*

`RTC_GPIO_MODE_INPUT_ONLY`

Pad input

`RTC_GPIO_MODE_OUTPUT_ONLY`

Pad output

`RTC_GPIO_MODE_INPUT_OUTPUT`

Pad pull input + output

`RTC_GPIO_MODE_DISABLED`

Pad (output + input) disable

## 2.4.4 I2C

An I2C (Inter-Integrated Circuit) bus can be used for communication with several external devices connected to the same bus as ESP32. There are two I2C controllers on board of the ESP32, each of which can be set to master mode or slave mode.

### Overview

The following sections will walk you through typical steps to configure and operate the I2C driver:

1. *Configure Driver* - select driver's parameters like master or slave mode, set specific GPIO pins to act as SDA and SCL, set the clock speed, etc.
2. *Install Driver*- activate driver in master or slave mode to operate on one of the two I2C controllers available on ESP32.
3. *Run Communication*:
  - a) *Master Mode* - run communication acting as a master
  - b) *Slave Mode* - get slave responding to messages from the master
4. *Interrupt Handling* - configure and service I2C interrupts.
5. *Going Beyond Defaults* - adjust timing, pin configuration and other parameters of the I2C communication.
6. *Error Handling* - how to recognize and handle driver configuration and communication errors.
7. *Delete Driver*- on communication end to free resources used by the I2C driver.

The top level identification of an I2C driver is one of the two port numbers selected from `i2c_port_t`. The mode of operation for a given port is provided during driver configuration by selecting either “master” or “slave” from `i2c_mode_t`.

## Configure Driver

The first step to establishing I2C communication is to configure the driver. This is done by setting several parameters contained in `i2c_config_t` structure:

- I2C **operation mode** - select either slave or master from `i2c_opmode_t`
- Settings of the **communication pins**:
  - GPIO pin numbers assigned to the SDA and SCL signals
  - Whether to enable ESP32's internal pull up for respective pins
- I2C **clock speed**, if this configuration concerns the master mode
- If this configuration concerns the slave mode:
  - Whether **10 bit address mode** should be enabled
  - The **slave address**

Then, to initialize configuration for a given I2C port, call function `i2c_param_config()` with the port number and `i2c_config_t` structure as the function call parameters.

At this stage `i2c_param_config()` also sets “behind the scenes” couple of other I2C configuration parameters to commonly used default values. To check what are the values and how to change them, see *Going Beyond Defaults*.

## Install Driver

Having the configuration initialized, the next step is to install the I2C driver by calling `i2c_driver_install()`. This function call requires the following parameters:

- The port number, one of the two ports available, selected from `i2c_port_t`
- The operation mode, slave or master selected from `i2c_opmode_t`
- Sizes of buffers that will be allocated for sending and receiving data **in the slave mode**
- Flags used to allocate the interrupt

## Run Communication

With the I2C driver installed, ESP32 is ready to communicate with other I2C devices. Programming of communication depends on whether selected I2C port operates in a master or a slave mode.

### Master Mode

ESP32's I2C port working in the master mode is responsible for establishing communication with slave I2C devices and sending commands to trigger actions by slaves, like doing a measurement and sending back a result.

To organize this process the driver provides a container, called a “command link”, that should be populated with a sequence of commands and then passed to the I2C controller for execution.

### Master Write

An example of building a command link for I2C master sending *n* bytes to slave is shown below:

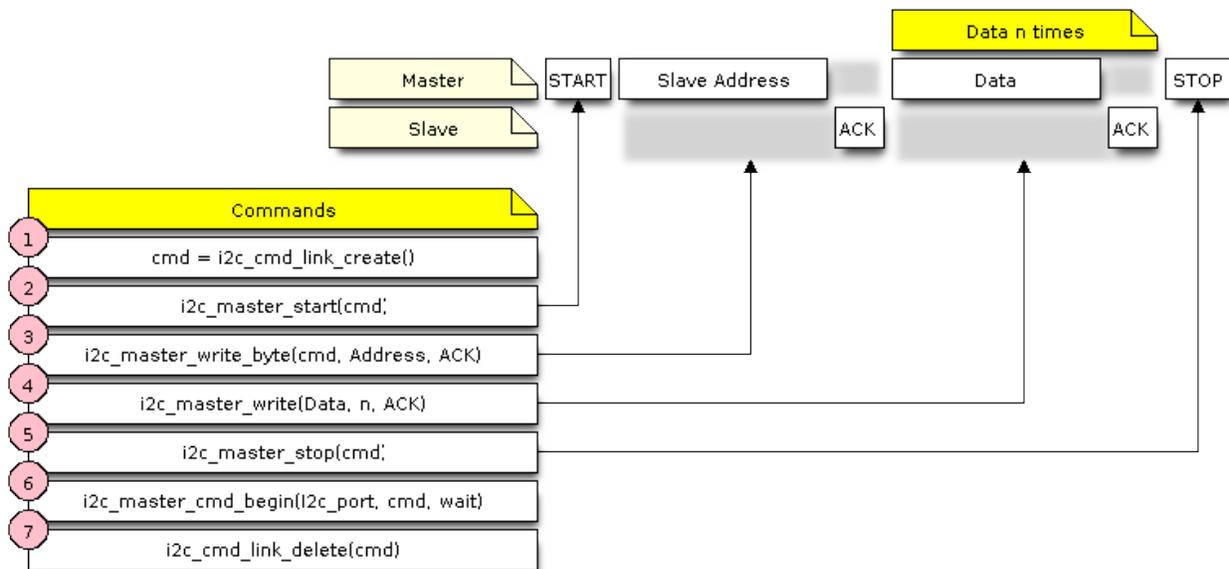


Fig. 3: I2C command link - master write example

The following describes how the command link for a “master write” is set up and what comes inside:

1. The first step is to create a command link with `i2c_cmd_link_create()`.

Then the command link is populated with series of data to be sent to the slave:

2. **Start bit** - `i2c_master_start()`
3. Single byte **slave address** - `i2c_master_write_byte()`. The address is provided as an argument of this function call.
4. One or more bytes of **data** as an argument of `i2c_master_write()`.
5. **Stop bit** - `i2c_master_stop()`

Both `i2c_master_write_byte()` and `i2c_master_write()` commands have additional argument defining whether slave should **acknowledge** received data or not.

6. Execution of command link by I2C controller is triggered by calling `i2c_master_cmd_begin()`.

- As the last step, after sending of the commands is finished, the resources used by the command link are released by calling `i2c_cmd_link_delete()`.

### Master Read

There is a similar sequence of steps for the master to read the data from a slave.

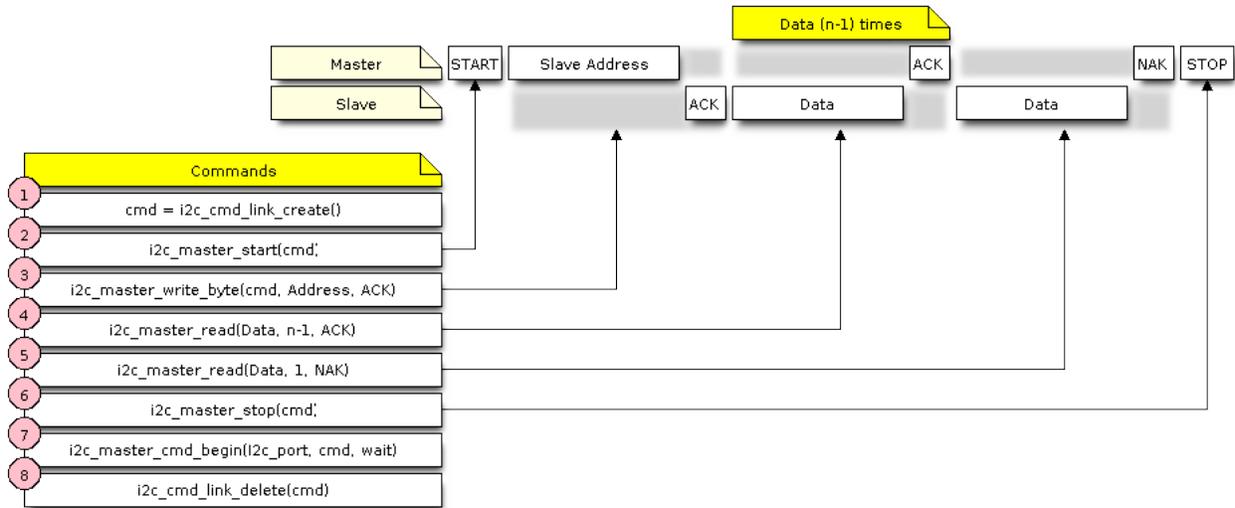


Fig. 4: I2C command link - master read example

When reading the data, instead of “`i2c_master_read...`”, the command link is populated with `i2c_master_read_byte()` and / or `i2c_master_read()`. Also, the last read is configured for not providing an acknowledge by the master.

### Master Write or Read?

After sending a slave’s address, see step 3 on pictures above, the master either writes to or reads from the slave. The information what the master will actually do is hidden in the least significant bit of the slave’s address.

Therefore the command link instructing the slave that the master will write the data contains the address like `(ESP_SLAVE_ADDR << 1) | I2C_MASTER_WRITE` and looks as follows:

```
i2c_master_write_byte(cmd, (ESP_SLAVE_ADDR << 1) | I2C_MASTER_WRITE, ACK_CHECK_EN)
```

By similar token the command link to read from the slave looks as follows:

```
i2c_master_write_byte(cmd, (ESP_SLAVE_ADDR << 1) | I2C_MASTER_READ, ACK_CHECK_EN)
```

### Slave Mode

The API provides functions to read and write data by the slave - \* `i2c_slave_read_buffer()` and `i2c_slave_write_buffer()`. An example of using these functions is provided in `peripherals/i2c`.

### Interrupt Handling

To register an interrupt handler, call function `i2c_isr_register()`, to delete the handler call `i2c_isr_free()`. Description of interrupts triggered by I2C controller is provided in the [ESP32 Technical Reference Manual \(PDF\)](#).

## Going Beyond Defaults

There are couple of I2C communication parameters setup during driver configuration (when calling `i2c_param_config()`, see *Configure Driver*), to some default commonly used values. Some parameters are also already configured in registers of the I2C controller. These parameters can be changed to user defined values by calling dedicated functions:

- Period of SCL pulses being high and low - `i2c_set_period()`
- SCL and SDA signal timing used during generation of start / stop signals - `i2c_set_start_timing()` / `i2c_set_stop_timing()`
- Timing relationship between SCL and SDA signals when sampling by slave, as well as when transmitting by master - `i2c_set_data_timing()`
- I2C timeout - `i2c_set_timeout()`

---

**Note:** The timing values are defined in APB clock cycles. The frequency of APB is specified in `I2C_APB_CLK_FREQ`.

---

- What bit, LSB or MSB, is transmitted / received first - `i2c_set_data_mode()` selectable out of modes defined in `i2c_trans_mode_t`

Each one of the above functions has a `_get_` counterpart to check the currently set value.

To see the default values of parameters setup during driver configuration, please refer to file `driver/i2c.c` looking up defines with `_DEFAULT` suffix.

With function `i2c_set_pin()` it is also possible to select different SDA and SCL pins and alter configuration of pull ups, changing what has been already entered with `i2c_param_config()`.

---

**Note:** ESP32's internal pull ups are in the range of some tens of kOhm, and as such in most cases insufficient for use as I2C pull ups by themselves. We suggest to add external pull ups as well, with values as described in the I2C standard.

---

## Error Handling

Most of driver's function return the `ESP_OK` on successful completion or a specific error code on a failure. It is a good practice to always check the returned values and implement the error handling. The driver is also printing out log messages, when e.g. checking the correctness of entered configuration, that contain explanation of errors. For details please refer to file `driver/i2c.c` looking up defines with `_ERR_STR` suffix.

Use dedicated interrupts to capture communication failures. For instance there is `I2C_TIME_OUT_INT` interrupt triggered when I2C takes too long to receive data. See *Interrupt Handling* for related information.

To reset internal hardware buffers in case of communication failure, you can use `i2c_reset_tx_fifo()` and `i2c_reset_rx_fifo()`.

## Delete Driver

If the I2C communication is established with `i2c_driver_install()` for some specific period of time and then not required, the driver may be removed to free allocated resources by calling `i2c_driver_delete()`.

## Application Example

I2C master and slave example: [peripherals/i2c](#).

## API Reference

### Header File

- [driver/include/driver/i2c.h](#)

### Functions

`esp_err_t i2c_driver_install(i2c_port_t i2c_num, i2c_mode_t mode, size_t slv_rx_buf_len, size_t slv_tx_buf_len, int intr_alloc_flags)`

I2C driver install.

**Note** Only slave mode will use this value, driver will ignore this value in master mode.

**Note** Only slave mode will use this value, driver will ignore this value in master mode.

**Note** In master mode, if the cache is likely to be disabled(such as write flash) and the slave is time-sensitive, `ESP_INTR_FLAG_IRAM` is suggested to be used. In this case, please use the memory allocated from internal RAM in i2c read and write function, because we can not access the psram(if psram is enabled) in interrupt handle function when cache is disabled.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_FAIL` Driver install error

#### Parameters

- `i2c_num`: I2C port number
- `mode`: I2C mode( master or slave )
- `slv_rx_buf_len`: receiving buffer size for slave mode

#### Parameters

- `slv_tx_buf_len`: sending buffer size for slave mode

#### Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.

`esp_err_t i2c_driver_delete(i2c_port_t i2c_num)`

I2C driver delete.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `i2c_num`: I2C port number

`esp_err_t i2c_param_config` (`i2c_port_t i2c_num`, `const i2c_config_t *i2c_conf`)  
I2C parameter initialization.

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `i2c_conf`: pointer to I2C parameter settings

`esp_err_t i2c_reset_tx_fifo` (`i2c_port_t i2c_num`)  
reset I2C tx hardware fifo

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number

`esp_err_t i2c_reset_rx_fifo` (`i2c_port_t i2c_num`)  
reset I2C rx fifo

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number

`esp_err_t i2c_isr_register` (`i2c_port_t i2c_num`, `void (*fn)`) `void *`  
`, void *arg`, `int intr_alloc_flags`, `intr_handle_t *handle` I2C isr handler register.

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `fn`: isr handler function
- `arg`: parameter for isr handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `handle`: handle return from `esp_intr_alloc`.

`esp_err_t i2c_isr_free` (*intr\_handle\_t handle*)  
to delete and free I2C isr.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- handle: handle of isr.

`esp_err_t i2c_set_pin` (*i2c\_port\_t i2c\_num*, *int sda\_io\_num*, *int scl\_io\_num*, *gpio\_pullup\_t sda\_pullup\_en*,  
*gpio\_pullup\_t scl\_pullup\_en*, *i2c\_mode\_t mode*)  
Configure GPIO signal for I2C sck and sda.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- i2c\_num: I2C port number
- sda\_io\_num: GPIO number for I2C sda signal
- scl\_io\_num: GPIO number for I2C scl signal
- sda\_pullup\_en: Whether to enable the internal pullup for sda pin
- scl\_pullup\_en: Whether to enable the internal pullup for scl pin
- mode: I2C mode

*i2c\_cmd\_handle\_t i2c\_cmd\_link\_create* ()  
Create and init I2C command link.

**Note** Before we build I2C command link, we need to call `i2c_cmd_link_create()` to create a command link. After we finish sending the commands, we need to call `i2c_cmd_link_delete()` to release and return the resources.

**Return** i2c command link handler

`void i2c_cmd_link_delete` (*i2c\_cmd\_handle\_t cmd\_handle*)  
Free I2C command link.

**Note** Before we build I2C command link, we need to call `i2c_cmd_link_create()` to create a command link. After we finish sending the commands, we need to call `i2c_cmd_link_delete()` to release and return the resources.

#### Parameters

- cmd\_handle: I2C command handle

`esp_err_t i2c_master_start` (*i2c\_cmd\_handle\_t cmd\_handle*)  
Queue command for I2C master to generate a start signal.

**Note** Only call this function in I2C master mode Call `i2c_master_cmd_begin()` to send all queued commands

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- cmd\_handle: I2C cmd link

esp\_err\_t **i2c\_master\_write\_byte** (*i2c\_cmd\_handle\_t* cmd\_handle, uint8\_t data, bool ack\_en)  
Queue command for I2C master to write one byte to I2C bus.

**Note** Only call this function in I2C master mode Call i2c\_master\_cmd\_begin() to send all queued commands

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- cmd\_handle: I2C cmd link
- data: I2C one byte command to write to bus
- ack\_en: enable ack check for master

esp\_err\_t **i2c\_master\_write** (*i2c\_cmd\_handle\_t* cmd\_handle, uint8\_t \*data, size\_t data\_len, bool ack\_en)  
Queue command for I2C master to write buffer to I2C bus.

**Note** Only call this function in I2C master mode Call i2c\_master\_cmd\_begin() to send all queued commands

**Note** If the psram is enabled and intr\_flag is ESP\_INTR\_FLAG\_IRAM, please use the memory allocated from internal RAM.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- cmd\_handle: I2C cmd link
- data: data to send

**Parameters**

- data\_len: data length
- ack\_en: enable ack check for master

esp\_err\_t **i2c\_master\_read\_byte** (*i2c\_cmd\_handle\_t* cmd\_handle, uint8\_t \*data, *i2c\_ack\_type\_t* ack)  
Queue command for I2C master to read one byte from I2C bus.

**Note** Only call this function in I2C master mode Call i2c\_master\_cmd\_begin() to send all queued commands

**Note** If the psram is enabled and intr\_flag is ESP\_INTR\_FLAG\_IRAM, please use the memory allocated from internal RAM.

**Return**

- ESP\_OK Success

- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- cmd\_handle: I2C cmd link
- data: pointer accept the data byte

**Parameters**

- ack: ack value for read command

esp\_err\_t **i2c\_master\_read**(*i2c\_cmd\_handle\_t cmd\_handle*, uint8\_t \**data*, size\_t *data\_len*, *i2c\_ack\_type\_t ack*)

Queue command for I2C master to read data from I2C bus.

**Note** Only call this function in I2C master mode Call i2c\_master\_cmd\_begin() to send all queued commands

**Note** If the psram is enabled and intr\_flag is ESP\_INTR\_FLAG\_IRAM, please use the memory allocated from internal RAM.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- cmd\_handle: I2C cmd link
- data: data buffer to accept the data from bus

**Parameters**

- data\_len: read data length
- ack: ack value for read command

esp\_err\_t **i2c\_master\_stop**(*i2c\_cmd\_handle\_t cmd\_handle*)

Queue command for I2C master to generate a stop signal.

**Note** Only call this function in I2C master mode Call i2c\_master\_cmd\_begin() to send all queued commands

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- cmd\_handle: I2C cmd link

esp\_err\_t **i2c\_master\_cmd\_begin**(*i2c\_port\_t i2c\_num*, *i2c\_cmd\_handle\_t cmd\_handle*, TickType\_t *ticks\_to\_wait*)

I2C master send queued commands. This function will trigger sending all queued commands. The task will be blocked until all the commands have been sent out. The I2C APIs are not thread-safe, if you want to use one I2C port in different tasks, you need to take care of the multi-thread issue.

**Note** Only call this function in I2C master mode

**Return**

- ESP\_OK Success

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_FAIL Sending command error, slave doesn't ACK the transfer.
- ESP\_ERR\_INVALID\_STATE I2C driver not installed or not in master mode.
- ESP\_ERR\_TIMEOUT Operation timeout because the bus is busy.

#### Parameters

- `i2c_num`: I2C port number
- `cmd_handle`: I2C command handler
- `ticks_to_wait`: maximum wait ticks.

int **i2c\_slave\_write\_buffer** (*i2c\_port\_t* `i2c_num`, uint8\_t \*`data`, int `size`, TickType\_t `ticks_to_wait`)  
I2C slave write data to internal ringbuffer, when tx fifo empty, isr will fill the hardware fifo from the internal ringbuffer.

**Note** Only call this function in I2C slave mode

#### Return

- ESP\_FAIL(-1) Parameter error
- Others(>=0) The number of data bytes that pushed to the I2C slave buffer.

#### Parameters

- `i2c_num`: I2C port number
- `data`: data pointer to write into internal buffer
- `size`: data size
- `ticks_to_wait`: Maximum waiting ticks

int **i2c\_slave\_read\_buffer** (*i2c\_port\_t* `i2c_num`, uint8\_t \*`data`, size\_t `max_size`, TickType\_t `ticks_to_wait`)  
I2C slave read data from internal buffer. When I2C slave receive data, isr will copy received data from hardware rx fifo to internal ringbuffer. Then users can read from internal ringbuffer.

**Note** Only call this function in I2C slave mode

#### Return

- ESP\_FAIL(-1) Parameter error
- Others(>=0) The number of data bytes that read from I2C slave buffer.

#### Parameters

- `i2c_num`: I2C port number
- `data`: data pointer to write into internal buffer
- `max_size`: Maximum data size to read
- `ticks_to_wait`: Maximum waiting ticks

esp\_err\_t **i2c\_set\_period** (*i2c\_port\_t* `i2c_num`, int `high_period`, int `low_period`)  
set I2C master clock period

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `high_period`: clock cycle number during SCL is high level, `high_period` is a 14 bit value
- `low_period`: clock cycle number during SCL is low level, `low_period` is a 14 bit value

`esp_err_t i2c_get_period(i2c_port_t i2c_num, int *high_period, int *low_period)`  
get I2C master clock period

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `high_period`: pointer to get clock cycle number during SCL is high level, will get a 14 bit value
- `low_period`: pointer to get clock cycle number during SCL is low level, will get a 14 bit value

`esp_err_t i2c_filter_enable(i2c_port_t i2c_num, uint8_t cyc_num)`  
enable hardware filter on I2C bus Sometimes the I2C bus is disturbed by high frequency noise(about 20ns), or the rising edge of the SCL clock is very slow, these may cause the master state machine broken. enable hardware filter can filter out high frequency interference and make the master more stable.

**Note** Enable filter will slow the SCL clock.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `cyc_num`: the APB cycles need to be filtered( $0 \leq \text{cyc\_num} \leq 7$ ). When the period of a pulse is less than `cyc_num * APB_cycle`, the I2C controller will ignore this pulse.

`esp_err_t i2c_filter_disable(i2c_port_t i2c_num)`  
disable filter on I2C bus

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number

`esp_err_t i2c_set_start_timing(i2c_port_t i2c_num, int setup_time, int hold_time)`  
set I2C master start signal timing

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `setup_time`: clock number between the falling-edge of SDA and rising-edge of SCL for start mark, it's a 10-bit value.
- `hold_time`: clock num between the falling-edge of SDA and falling-edge of SCL for start mark, it's a 10-bit value.

`esp_err_t i2c_get_start_timing(i2c_port_t i2c_num, int *setup_time, int *hold_time)`  
get I2C master start signal timing

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `setup_time`: pointer to get setup time
- `hold_time`: pointer to get hold time

`esp_err_t i2c_set_stop_timing(i2c_port_t i2c_num, int setup_time, int hold_time)`  
set I2C master stop signal timing

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `setup_time`: clock num between the rising-edge of SCL and the rising-edge of SDA, it's a 10-bit value.
- `hold_time`: clock number after the STOP bit's rising-edge, it's a 14-bit value.

`esp_err_t i2c_get_stop_timing(i2c_port_t i2c_num, int *setup_time, int *hold_time)`  
get I2C master stop signal timing

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `setup_time`: pointer to get setup time.

- `hold_time`: pointer to get hold time.

`esp_err_t i2c_set_data_timing(i2c_port_t i2c_num, int sample_time, int hold_time)`  
set I2C data signal timing

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `sample_time`: clock number I2C used to sample data on SDA after the rising-edge of SCL, it's a 10-bit value
- `hold_time`: clock number I2C used to hold the data after the falling-edge of SCL, it's a 10-bit value

`esp_err_t i2c_get_data_timing(i2c_port_t i2c_num, int *sample_time, int *hold_time)`  
get I2C data signal timing

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `sample_time`: pointer to get sample time
- `hold_time`: pointer to get hold time

`esp_err_t i2c_set_timeout(i2c_port_t i2c_num, int timeout)`  
set I2C timeout value

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number
- `timeout`: timeout value for I2C bus (unit: APB 80Mhz clock cycle)

`esp_err_t i2c_get_timeout(i2c_port_t i2c_num, int *timeout)`  
get I2C timeout value

**Return**

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

**Parameters**

- `i2c_num`: I2C port number

- `timeout`: pointer to get timeout value

`esp_err_t i2c_set_data_mode` (`i2c_port_t` `i2c_num`, `i2c_trans_mode_t` `tx_trans_mode`, `i2c_trans_mode_t` `rx_trans_mode`)  
set I2C data transfer mode

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `i2c_num`: I2C port number
- `tx_trans_mode`: I2C sending data mode
- `rx_trans_mode`: I2C receiving data mode

`esp_err_t i2c_get_data_mode` (`i2c_port_t` `i2c_num`, `i2c_trans_mode_t` `*tx_trans_mode`, `i2c_trans_mode_t` `*rx_trans_mode`)  
get I2C data transfer mode

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `i2c_num`: I2C port number
- `tx_trans_mode`: pointer to get I2C sending data mode
- `rx_trans_mode`: pointer to get I2C receiving data mode

## Structures

`struct i2c_config_t`  
I2C initialization parameters.

#### Public Members

`i2c_mode_t mode`  
I2C mode

`gpio_num_t sda_io_num`  
GPIO number for I2C sda signal

`gpio_pullup_t sda_pullup_en`  
Internal GPIO pull mode for I2C sda signal

`gpio_num_t scl_io_num`  
GPIO number for I2C scl signal

`gpio_pullup_t scl_pullup_en`  
Internal GPIO pull mode for I2C scl signal

`uint32_t clk_speed`  
I2C clock frequency for master mode, (no higher than 1MHz for now)

`uint8_t addr_10bit_en`  
I2C 10bit address mode enable for slave mode

`uint16_t slave_addr`  
I2C address for slave mode

## Macros

**I2C\_APB\_CLK\_FREQ**  
I2C source clock is APB clock, 80MHz

**I2C\_FIFO\_LEN**  
I2C hardware fifo length

## Type Definitions

**typedef void \*i2c\_cmd\_handle\_t**  
I2C command handle

## Enumerations

**enum i2c\_mode\_t**  
*Values:*

**I2C\_MODE\_SLAVE = 0**  
I2C slave mode

**I2C\_MODE\_MASTER**  
I2C master mode

**I2C\_MODE\_MAX**

**enum i2c\_rw\_t**  
*Values:*

**I2C\_MASTER\_WRITE = 0**  
I2C write data

**I2C\_MASTER\_READ**  
I2C read data

**enum i2c\_trans\_mode\_t**  
*Values:*

**I2C\_DATA\_MODE\_MSB\_FIRST = 0**  
I2C data msb first

**I2C\_DATA\_MODE\_LSB\_FIRST = 1**  
I2C data lsb first

**I2C\_DATA\_MODE\_MAX**

**enum i2c\_opmode\_t**  
*Values:*

**I2C\_CMD\_RESTART** = 0  
I2C restart command

**I2C\_CMD\_WRITE**  
I2C write command

**I2C\_CMD\_READ**  
I2C read command

**I2C\_CMD\_STOP**  
I2C stop command

**I2C\_CMD\_END**  
I2C end command

**enum i2c\_port\_t**

*Values:*

**I2C\_NUM\_0** = 0  
I2C port 0

**I2C\_NUM\_1**  
I2C port 1

**I2C\_NUM\_MAX**

**enum i2c\_addr\_mode\_t**

*Values:*

**I2C\_ADDR\_BIT\_7** = 0  
I2C 7bit address for slave mode

**I2C\_ADDR\_BIT\_10**  
I2C 10bit address for slave mode

**I2C\_ADDR\_BIT\_MAX**

**enum i2c\_ack\_type\_t**

*Values:*

**I2C\_MASTER\_ACK** = 0x0  
I2C ack for each byte read

**I2C\_MASTER\_NACK** = 0x1  
I2C nack for each byte read

**I2C\_MASTER\_LAST\_NACK** = 0x2  
I2C nack for the last byte

**I2C\_MASTER\_ACK\_MAX**

## 2.4.5 I2S

### Overview

ESP32 contains two I2S peripherals. These peripherals can be configured to input and output sample data via the I2S driver.

The I2S peripheral supports DMA meaning it can stream sample data without requiring each sample to be read or written by the CPU.

I2S output can also be routed directly to the Digital/Analog Converter output channels (GPIO 25 & GPIO 26) to produce analog output directly, rather than via an external I2S codec.

**Note:** For high accuracy clock applications, APLL clock source can be used with `.use_apll = 1` and ESP32 will automatic caculate APLL parameter.

## Application Example

A full I2S example is available in esp-idf: `peripherals/i2s`.

Short example of I2S configuration:

```
#include "driver/i2s.h"
#include "freertos/queue.h"

static const int i2s_num = 0; // i2s port number

static const i2s_config_t i2s_config = {
    .mode = I2S_MODE_MASTER | I2S_MODE_TX,
    .sample_rate = 44100,
    .bits_per_sample = 16,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = 0, // default interrupt priority
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = 0
};

static const i2s_pin_config_t pin_config = {
    .bck_io_num = 26,
    .ws_io_num = 25,
    .data_out_num = 22,
    .data_in_num = I2S_PIN_NO_CHANGE
};

...

i2s_driver_install(i2s_num, &i2s_config, 0, NULL); //install and start i2s_
↔driver

i2s_set_pin(i2s_num, &pin_config);

i2s_set_sample_rates(i2s_num, 22050); //set sample rates

i2s_driver_uninstall(i2s_num); //stop & destroy i2s driver
```

Short example configuring I2S to use internal DAC for analog output:

```
#include "driver/i2s.h"
#include "freertos/queue.h"

static const int i2s_num = 0; // i2s port number

static const i2s_config_t i2s_config = {
```

(continues on next page)

(continued from previous page)

```

        .mode = I2S_MODE_MASTER | I2S_MODE_TX | I2S_MODE_DAC_BUILT_IN,
        .sample_rate = 44100,
        .bits_per_sample = 16, /* the DAC module will only take the 8bits from MSB */
        .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
        .communication_format = I2S_COMM_FORMAT_I2S_MSB,
        .intr_alloc_flags = 0, // default interrupt priority
        .dma_buf_count = 8,
        .dma_buf_len = 64,
        .use_apll = 0
    };

    ...

    i2s_driver_install(i2s_num, &i2s_config, 0, NULL); //install and start i2s_
↳driver

    i2s_set_pin(i2s_num, NULL); //for internal DAC, this will enable both of the_
↳internal channels

    //You can call i2s_set_dac_mode to set built-in DAC output mode.
    //i2s_set_dac_mode(I2S_DAC_CHANNEL_BOTH_EN);

    i2s_set_sample_rates(i2s_num, 22050); //set sample rates

    i2s_driver_uninstall(i2s_num); //stop & destroy i2s driver

```

## API Reference

### Header File

- driver/include/driver/i2s.h

### Functions

`esp_err_t i2s_set_pin(i2s_port_t i2s_num, const i2s_pin_config_t *pin)`

Set I2S pin number.

Inside the pin configuration structure, set I2S\_PIN\_NO\_CHANGE for any pin where the current configuration should not be changed.

**Note** The I2S peripheral output signals can be connected to multiple GPIO pads. However, the I2S peripheral input signal can only be connected to one GPIO pad.

#### Parameters

- `i2s_num`: I2S\_NUM\_0 or I2S\_NUM\_1
- `pin`: I2S Pin structure, or NULL to set 2-channel 8-bit internal DAC pin configuration (GPIO25 & GPIO26)

**Note** if `*pin` is set as NULL, this function will initialize both of the built-in DAC channels by default. if you don't want this to happen and you want to initialize only one of the DAC channels, you can call `i2s_set_dac_mode` instead.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_FAIL IO error

esp\_err\_t **i2s\_set\_dac\_mode** (*i2s\_dac\_mode\_t* *dac\_mode*)

Set I2S dac mode, I2S built-in DAC is disabled by default.

**Note** Built-in DAC functions are only supported on I2S0 for current ESP32 chip. If either of the built-in DAC channel are enabled, the other one can not be used as RTC DAC function at the same time.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- *dac\_mode*: DAC mode configurations - see *i2s\_dac\_mode\_t*

esp\_err\_t **i2s\_driver\_install** (*i2s\_port\_t* *i2s\_num*, **const** *i2s\_config\_t* \**i2s\_config*, int *queue\_size*, void \**i2s\_queue*)

Install and start I2S driver.

This function must be called before any I2S driver read/write operations.

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1
- *i2s\_config*: I2S configurations - see *i2s\_config\_t* struct
- *queue\_size*: I2S event queue size/depth.
- *i2s\_queue*: I2S event queue handle, if set NULL, driver will not use an event queue.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NO\_MEM Out of memory

esp\_err\_t **i2s\_driver\_uninstall** (*i2s\_port\_t* *i2s\_num*)

Uninstall I2S driver.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1

int **i2s\_write\_bytes** (*i2s\_port\_t* *i2s\_num*, **const** void \**src*, size\_t *size*, TickType\_t *ticks\_to\_wait*)

Write data to I2S DMA transmit buffer.

This function is deprecated. Use 'i2s\_write' instead. This definition will be removed in a future release.

#### Return

- The amount of bytes written, if timeout, the result will be less than the size passed in.
- ESP\_FAIL Parameter error

`esp_err_t i2s_write(i2s_port_t i2s_num, const void *src, size_t size, size_t *bytes_written, TickType_t ticks_to_wait)`  
Write data to I2S DMA transmit buffer.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `i2s_num`: I2S\_NUM\_0, I2S\_NUM\_1
- `src`: Source address to write from
- `size`: Size of data in bytes
- `bytes_written`: Number of bytes written, if timeout, the result will be less than the size passed in.
- `ticks_to_wait`: TX buffer wait timeout in RTOS ticks. If this many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass `portMAX_DELAY` for no timeout.

`esp_err_t i2s_write_expand(i2s_port_t i2s_num, const void *src, size_t size, size_t src_bits, size_t aim_bits, size_t *bytes_written, TickType_t ticks_to_wait)`

Write data to I2S DMA transmit buffer while expanding the number of bits per sample. For example, expanding 16-bit PCM to 32-bit PCM.

Format of the data in source buffer is determined by the I2S configuration (see `i2s_config_t`).

#### Parameters

- `i2s_num`: I2S\_NUM\_0, I2S\_NUM\_1
- `src`: Source address to write from
- `size`: Size of data in bytes
- `src_bits`: Source audio bit
- `aim_bits`: Bit wanted, no more than 32, and must be greater than `src_bits`
- `bytes_written`: Number of bytes written, if timeout, the result will be less than the size passed in.
- `ticks_to_wait`: TX buffer wait timeout in RTOS ticks. If this many ticks pass without space becoming available in the DMA transmit buffer, then the function will return (note that if the data is written to the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass `portMAX_DELAY` for no timeout.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

int **i2s\_read\_bytes** (*i2s\_port\_t* *i2s\_num*, void \**dest*, size\_t *size*, TickType\_t *ticks\_to\_wait*)  
Read data from I2S DMA receive buffer.

This function is deprecated. Use 'i2s\_read' instead. This definition will be removed in a future release.

#### Return

- The amount of bytes read, if timeout, bytes read will be less than the size passed in
- ESP\_FAIL Parameter error

esp\_err\_t **i2s\_read** (*i2s\_port\_t* *i2s\_num*, void \**dest*, size\_t *size*, size\_t \**bytes\_read*, TickType\_t *ticks\_to\_wait*)  
Read data from I2S DMA receive buffer.

**Note** If the built-in ADC mode is enabled, we should call `i2s_adc_start` and `i2s_adc_stop` around the whole reading process, to prevent the data getting corrupted.

**Note** If the built-in ADC mode is enabled, we should call `i2s_adc_start` and `i2s_adc_stop` around the whole reading process, to prevent the data getting corrupted.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1
- *dest*: Destination address to read into
- *size*: Size of data in bytes
- *bytes\_read*: Number of bytes read, if timeout, bytes read will be less than the size passed in.
- *ticks\_to\_wait*: RX buffer wait timeout in RTOS ticks. If this many ticks pass without bytes becoming available in the DMA receive buffer, then the function will return (note that if data is read from the DMA buffer in pieces, the overall operation may still take longer than this timeout.) Pass `portMAX_DELAY` for no timeout.

int **i2s\_push\_sample** (*i2s\_port\_t* *i2s\_num*, const void \**sample*, TickType\_t *ticks\_to\_wait*)  
Write a single sample to the I2S DMA TX buffer.

This function is deprecated. Use 'i2s\_write' instead. This definition will be removed in a future release.

#### Return

- Number of bytes successfully pushed to DMA buffer, will be either zero or the size of configured sample buffer (in bytes).
- ESP\_FAIL Parameter error

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1
- *sample*: Buffer to read data. Size of buffer (in bytes) = `bits_per_sample / 8`.
- *ticks\_to\_wait*: Timeout in RTOS ticks. If a sample is not available in the DMA buffer within this period, no data is read and function returns zero.

int **i2s\_pop\_sample** (*i2s\_port\_t* *i2s\_num*, void \**sample*, TickType\_t *ticks\_to\_wait*)

Read a single sample from the I2S DMA RX buffer.

This function is deprecated. Use 'i2s\_read' instead. This definition will be removed in a future release.

#### Return

- Number of bytes successfully read from DMA buffer, will be either zero or the size of configured sample buffer (in bytes).
- ESP\_FAIL Parameter error

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1
- *sample*: Buffer to write data. Size of buffer (in bytes) = bits\_per\_sample / 8.
- *ticks\_to\_wait*: Timeout in RTOS ticks. If a sample is not available in the DMA buffer within this period, no data is read and function returns zero.

esp\_err\_t **i2s\_set\_sample\_rates** (*i2s\_port\_t* *i2s\_num*, uint32\_t *rate*)

Set sample rate used for I2S RX and TX.

The bit clock rate is determined by the sample rate and *i2s\_config\_t* configuration parameters (number of channels, bits\_per\_sample).

$bit\_clock = rate * (number\ of\ channels) * bits\_per\_sample$

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NO\_MEM Out of memory

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1
- *rate*: I2S sample rate (ex: 8000, 44100...)

esp\_err\_t **i2s\_stop** (*i2s\_port\_t* *i2s\_num*)

Stop I2S driver.

Disables I2S TX/RX, until i2s\_start() is called.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1

esp\_err\_t **i2s\_start** (*i2s\_port\_t* *i2s\_num*)

Start I2S driver.

It is not necessary to call this function after i2s\_driver\_install() (it is started automatically), however it is necessary to call it after i2s\_stop().

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1

`esp_err_t i2s_zero_dma_buffer` (*i2s\_port\_t i2s\_num*)

Zero the contents of the TX DMA buffer.

Pushes zero-byte samples into the TX DMA buffer, until it is full.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1

`esp_err_t i2s_set_clk` (*i2s\_port\_t i2s\_num*, `uint32_t rate`, *i2s\_bits\_per\_sample\_t bits*, *i2s\_channel\_t ch*)

Set clock & bit width used for I2S RX and TX.

Similar to `i2s_set_sample_rates()`, but also sets bit width.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NO\_MEM Out of memory

**Parameters**

- *i2s\_num*: I2S\_NUM\_0, I2S\_NUM\_1
- *rate*: I2S sample rate (ex: 8000, 44100...)
- *bits*: I2S bit width (I2S\_BITS\_PER\_SAMPLE\_16BIT, I2S\_BITS\_PER\_SAMPLE\_24BIT, I2S\_BITS\_PER\_SAMPLE\_32BIT)
- *ch*: I2S channel, (I2S\_CHANNEL\_MONO, I2S\_CHANNEL\_STEREO)

`esp_err_t i2s_set_adc_mode` (*adc\_unit\_t adc\_unit*, *adc1\_channel\_t adc\_channel*)

Set built-in ADC mode for I2S DMA, this function will initialize ADC pad, and set ADC parameters.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- *adc\_unit*: SAR ADC unit index
- *adc\_channel*: ADC channel index

`esp_err_t i2s_adc_enable(i2s_port_t i2s_num)`  
Start to use I2S built-in ADC mode.

**Note** This function would acquire the lock of ADC to prevent the data getting corrupted during the I2S peripheral is being used to do fully continuous ADC sampling.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_INVALID\_STATE driver state error
- ESP\_FAIL Internal driver error

**Parameters**

- `i2s_num`: i2s port index

`esp_err_t i2s_adc_disable(i2s_port_t i2s_num)`  
Stop to use I2S built-in ADC mode.

**Note** This function would release the lock of ADC so that other tasks can use ADC.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_INVALID\_STATE driver state error

**Parameters**

- `i2s_num`: i2s port index

**Structures**

**struct i2s\_config\_t**  
I2S configuration parameters for `i2s_param_config` function.

**Public Members**

`i2s_mode_t mode`  
I2S work mode

int `sample_rate`  
I2S sample rate

`i2s_bits_per_sample_t bits_per_sample`  
I2S bits per sample

`i2s_channel_fmt_t channel_format`  
I2S channel format

`i2s_comm_format_t communication_format`  
I2S communication format

int **intr\_alloc\_flags**

Flags used to allocate the interrupt. One or multiple (ORred) ESP\_INTR\_FLAG\_\* values. See esp\_intr\_alloc.h for more info

int **dma\_buf\_count**

I2S DMA Buffer Count

int **dma\_buf\_len**

I2S DMA Buffer Length

int **use\_apll**

I2S using APLL as main I2S clock, enable it to get accurate clock

**struct i2s\_event\_t**

Event structure used in I2S event queue.

### Public Members

*i2s\_event\_type\_t* **type**

I2S event type

size\_t **size**

I2S data size for I2S\_DATA event

**struct i2s\_pin\_config\_t**

I2S pin number for i2s\_set\_pin.

### Public Members

int **bck\_io\_num**

BCK in out pin

int **ws\_io\_num**

WS in out pin

int **data\_out\_num**

DATA out pin

int **data\_in\_num**

DATA in pin

### Macros

**I2S\_PIN\_NO\_CHANGE**

Use in *i2s\_pin\_config\_t* for pins which should not be changed

### Type Definitions

**typedef** *intr\_handle\_t* **i2s\_isr\_handle\_t**

## Enumerations

**enum i2s\_bits\_per\_sample\_t**

I2S bit width per sample.

*Values:*

**I2S\_BITS\_PER\_SAMPLE\_8BIT = 8**

I2S bits per sample: 8-bits

**I2S\_BITS\_PER\_SAMPLE\_16BIT = 16**

I2S bits per sample: 16-bits

**I2S\_BITS\_PER\_SAMPLE\_24BIT = 24**

I2S bits per sample: 24-bits

**I2S\_BITS\_PER\_SAMPLE\_32BIT = 32**

I2S bits per sample: 32-bits

**enum i2s\_channel\_t**

I2S channel.

*Values:*

**I2S\_CHANNEL\_MONO = 1**

I2S 1 channel (mono)

**I2S\_CHANNEL\_STEREO = 2**

I2S 2 channel (stereo)

**enum i2s\_comm\_format\_t**

I2S communication standard format.

*Values:*

**I2S\_COMM\_FORMAT\_I2S = 0x01**

I2S communication format I2S

**I2S\_COMM\_FORMAT\_I2S\_MSB = 0x02**

I2S format MSB

**I2S\_COMM\_FORMAT\_I2S\_LSB = 0x04**

I2S format LSB

**I2S\_COMM\_FORMAT\_PCM = 0x08**

I2S communication format PCM

**I2S\_COMM\_FORMAT\_PCM\_SHORT = 0x10**

PCM Short

**I2S\_COMM\_FORMAT\_PCM\_LONG = 0x20**

PCM Long

**enum i2s\_channel\_fmt\_t**

I2S channel format type.

*Values:*

**I2S\_CHANNEL\_FMT\_RIGHT\_LEFT = 0x00**

**I2S\_CHANNEL\_FMT\_ALL\_RIGHT**

**I2S\_CHANNEL\_FMT\_ALL\_LEFT**

**I2S\_CHANNEL\_FMT\_ONLY\_RIGHT**

**I2S\_CHANNEL\_FMT\_ONLY\_LEFT**

**enum pdm\_sample\_rate\_ratio\_t**  
PDM sample rate ratio, measured in Hz.

*Values:*

**PDM\_SAMPLE\_RATE\_RATIO\_64**

**PDM\_SAMPLE\_RATE\_RATIO\_128**

**enum pdm\_pcm\_conv\_t**  
PDM PCM convter enable/disable.

*Values:*

**PDM\_PCM\_CONV\_ENABLE**

**PDM\_PCM\_CONV\_DISABLE**

**enum i2s\_port\_t**  
I2S Peripheral, 0 & 1.

*Values:*

**I2S\_NUM\_0** = 0x0  
I2S 0

**I2S\_NUM\_1** = 0x1  
I2S 1

**I2S\_NUM\_MAX**

**enum i2s\_mode\_t**  
I2S Mode, default is I2S\_MODE\_MASTER | I2S\_MODE\_TX.

**Note** PDM and built-in DAC functions are only supported on I2S0 for current ESP32 chip.

*Values:*

**I2S\_MODE\_MASTER** = 1

**I2S\_MODE\_SLAVE** = 2

**I2S\_MODE\_TX** = 4

**I2S\_MODE\_RX** = 8

**I2S\_MODE\_DAC\_BUILT\_IN** = 16  
Output I2S data to built-in DAC, no matter the data format is 16bit or 32 bit, the DAC module will only take the 8bits from MSB

**I2S\_MODE\_ADC\_BUILT\_IN** = 32  
Input I2S data from built-in ADC, each data can be 12-bit width at most

**I2S\_MODE\_PDM** = 64

**enum i2s\_event\_type\_t**  
I2S event types.

*Values:*

**I2S\_EVENT\_DMA\_ERROR**

**I2S\_EVENT\_TX\_DONE**  
I2S DMA finish sent 1 buffer

**I2S\_EVENT\_RX\_DONE**  
I2S DMA finish received 1 buffer

**I2S\_EVENT\_MAX**  
I2S event max index

**enum i2s\_dac\_mode\_t**  
I2S DAC mode for `i2s_set_dac_mode`.

**Note** PDM and built-in DAC functions are only supported on I2S0 for current ESP32 chip.

*Values:*

**I2S\_DAC\_CHANNEL\_DISABLE = 0**  
Disable I2S built-in DAC signals

**I2S\_DAC\_CHANNEL\_RIGHT\_EN = 1**  
Enable I2S built-in DAC right channel, maps to DAC channel 1 on GPIO25

**I2S\_DAC\_CHANNEL\_LEFT\_EN = 2**  
Enable I2S built-in DAC left channel, maps to DAC channel 2 on GPIO26

**I2S\_DAC\_CHANNEL\_BOTH\_EN = 0x3**  
Enable both of the I2S built-in DAC channels.

**I2S\_DAC\_CHANNEL\_MAX = 0x4**  
I2S built-in DAC mode max index

## 2.4.6 LED Control

### Introduction

The LED control (LEDC) module is primarily designed to control the intensity of LEDs, although it can be used to generate PWM signals for other purposes as well. It has 16 channels which can generate independent waveforms, that can be used to drive e.g. RGB LED devices.

Half of all LEDC's channels provide high speed mode of operation. This mode offers implemented in hardware, automatic and glitch free change of PWM duty cycle. The other half of channels operate in a low speed mode, where the moment of change depends on the application software. Each group of channels is also able to use different clock sources but this feature is not implemented in the API.

The PWM controller also has the ability to automatically increase or decrease the duty cycle gradually, allowing for fades without any processor interference.

### Functionality Overview

Getting LEDC to work on specific channel in either *high or low speed mode* is done in three steps:

1. *Configure Timer* to determine PWM signal's frequency and the a number (resolution of duty range).
2. *Configure Channel* by associating it with the timer and GPIO to output the PWM signal.
3. *Change PWM Signal* that drives the output to change LED's intensity. This may be done under full control by software or with help of hardware fading functions.

In an optional step it is also possible to set up an interrupt on the fade end.

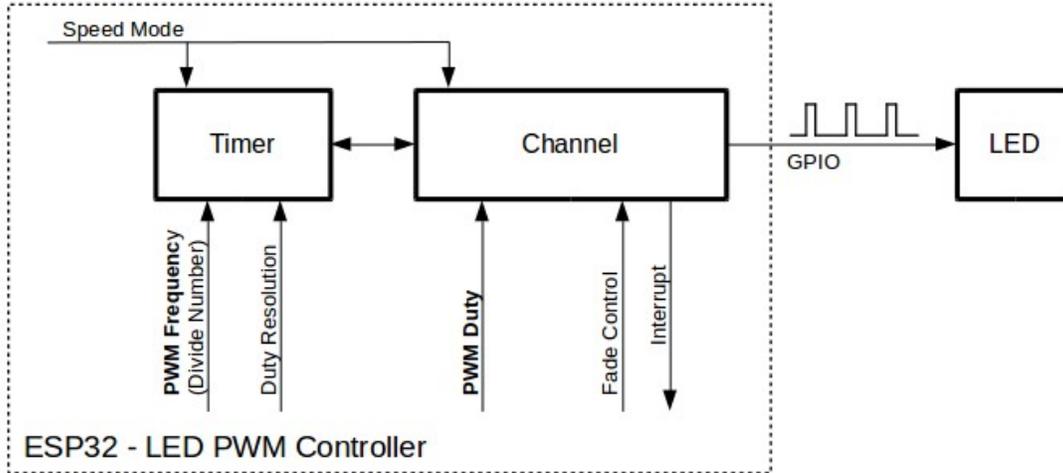


Fig. 5: Key Settings of LED PWM Controller's API

### Configure Timer

Setting of the timer is done by calling function `ledc_timer_config()`. This function should be provided with a data structure `ledc_timer_config_t` that contains the following configuration settings:

- The timer number `ledc_timer_t` and a speed mode `ledc_mode_t`.
- The PWM signal's frequency and resolution of PWM's duty value changes.

The frequency and the duty resolution are interdependent. The higher the PWM frequency, the lower duty resolution is available and vice versa. This relationship may become important, if you are planning to use this API for purposes other than changing intensity of LEDs. Check section *Supported Range of Frequency and Duty Resolution* for more details.

### Configure Channel

Having set up the timer, the next step is to configure selected channel (one out of `ledc_channel_t`). This is done by calling function `ledc_channel_config()`.

In similar way, like with the timer configuration, the channel setup function should be provided with specific structure `ledc_channel_config_t`, that contains channel's configuration parameters.

At this point channel should become operational and start generating PWM signal of frequency determined by the timer settings and the duty on selected GPIO, as configured in `ledc_channel_config_t`. The channel operation / the signal generation may be suspended at any time by calling function `ledc_stop()`.

### Change PWM Signal

Once the channel is operational and generating the PWM signal of constant duty and frequency, there are a couple of ways to change this signal. When driving LEDs we are changing primarily the duty to vary the light intensity. See the

two section below how to change the duty by software or with hardware fading. If required, we can change signal's frequency as well and this is covered in section *Change PWM Frequency*.

### Change PWM Duty by Software

Setting of the duty is done by first calling dedicated function `ledc_set_duty()` and then calling `ledc_update_duty()` to make the change effective. To check the value currently set, there is a corresponding `_get_` function `ledc_get_duty()`.

Another way to set the duty, and some other channel parameters as well, is by calling `ledc_channel_config()` discussed in the previous section.

The range of the duty value entered into functions depends on selected `duty_resolution` and should be from 0 to  $(2^{**} \text{duty\_resolution}) - 1$ . For example, if selected duty resolution is 10, then the duty range is from 0 to 1023. This provides the resolution of ~0.1%.

### Change PWM Duty with Hardware Fading

The LEDC hardware provides the means to gradually fade from one duty value to another. To use this functionality first enable fading with `ledc_fade_func_install()`. Then configure it by calling one of available fading functions:

- `ledc_set_fade_with_time()`
- `ledc_set_fade_with_step()`
- `ledc_set_fade()`

Finally start fading with `ledc_fade_start()`.

If not required anymore, fading and associated interrupt may be disabled with `ledc_fade_func_uninstall()`.

### Change PWM Frequency

The LEDC API provides several means to change the PWM frequency “on the fly”.

- One of options is to call `ledc_set_freq()`. There is a corresponding function `ledc_get_freq()` to check what frequency is currently set.
- Another option to change the frequency, and the duty resolution as well, is by calling `ledc_bind_channel_timer()` to bind other timer to the channel.
- Finally the channel's timer may be changed by calling `ledc_channel_config()`.

### More Control Over PWM

There are couple of lower level timer specific functions, that may be used to provide additional means to change the 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 “clean” start up of a timer after is it configured.

## Use Interrupts

When configuring a LEDC channel, one of parameters selected within `ledc_channel_config_t` is `ledc_intr_type_t` and allows to enable an interrupt on fade completion.

Registration of a handler to service this interrupt is done by calling `ledc_isr_register()`.

## LEDC High and Low Speed Mode

Out of the total 8 timers and 16 channels available in the LED PWM Controller, half of them are dedicated to operate in the high speed mode and the other half in the low speed mode. Selection of the low or high speed “capable” timer or the channel is done with parameter `ledc_mode_t` that is present in applicable function calls.

The advantage of the high speed mode is h/w supported, glitch-free changeover of the timer settings. This means that if the timer settings are modified, the changes will be applied automatically after the next overflow interrupt of the timer. In contrast, when updating the low-speed timer, the change of settings should be specifically triggered by software. The LEDC API is doing it “behind the scenes”, e.g. when `ledc_timer_config()` or `ledc_timer_set()` is called.

For additional details regarding speed modes please refer to [ESP32 Technical Reference Manual \(PDF\)](#). Note that support for `SLOW_CLOCK` mentioned in this manual is not implemented in the LEDC API.

## Supported Range of Frequency and Duty Resolution

The LED PWM Controller is designed primarily to drive LEDs and provides wide resolution of PWM duty settings. For instance for the PWM frequency at 5 kHz, the maximum duty resolution is 13 bits. It means that the duty may be set anywhere from 0 to 100% with resolution of ~0.012% ( $13 \times 2 = 8192$  discrete levels of the LED intensity).

The LEDC may be used for providing signals at much higher frequencies to clock other devices, e.g. a digital camera module. In such a case the maximum available frequency is 40 MHz with duty resolution of 1 bit. This means that duty is fixed at 50% and cannot be adjusted.

The API is designed to report an error when trying to set a frequency and a duty resolution that is out of the range of LEDC’s hardware. For example, an attempt to set the frequency at 20 MHz and the duty resolution of 3 bits will result in the following error reported on a serial monitor:

```
E (196) ledc: requested frequency and duty resolution can not be achieved, try
↪reducing freq_hz or duty_resolution. div_param=128
```

In such a case either the duty resolution or the frequency should be reduced. For example setting the duty resolution at 2 will resolve this issue and provide possibility to set the duty with 25% steps, i.e. at 25%, 50% or 75%.

The LEDC API will also capture and report an attempt to configure frequency / duty resolution combination that is below the supported minimum, e.g.:

```
E (196) ledc: requested frequency and duty resolution can not be achieved, try
↪increasing freq_hz or duty_resolution. div_param=128000000
```

Setting of the duty resolution is normally done using `ledc_timer_bit_t`. This enumeration covers the range from 10 to 15 bits. If a smaller duty resolution is required (below 10 down to 1), enter the equivalent numeric values directly.

## Application Example

The LEDC change duty cycle and fading control example: [peripherals/ledc](#).

## API Reference

### Header File

- `driver/include/driver/ledc.h`

### Functions

`esp_err_t ledc_channel_config(const ledc_channel_config_t *ledc_conf)`

LEDC channel configuration Configure LEDC channel with the given channel/output gpio\_num/interrupt/source timer/frequency(Hz)/LEDC duty resolution.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `ledc_conf`: Pointer of LEDC channel configure struct

`esp_err_t ledc_timer_config(const ledc_timer_config_t *timer_conf)`

LEDC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty\_resolution.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_FAIL Can not find a proper pre-divider number base on the given frequency and the current duty\_resolution.

#### Parameters

- `timer_conf`: Pointer of LEDC timer configure struct

`esp_err_t ledc_update_duty(ledc_mode_t speed_mode, ledc_channel_t channel)`

LEDC update channel parameters Call this function to activate the LEDC updated parameters. After `ledc_set_duty`, `ledc_set_fade`, we need to call this function to update the settings.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `speed_mode`: Select the LEDC speed\_mode, high-speed mode and low-speed mode,
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`

`esp_err_t ledc_stop(ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t idle_level)`

LEDC stop. Disable LEDC output, and set idle level.

#### Return

- ESP\_OK Success

- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `idle_level`: Set output idle level after LEDC stops.

`esp_err_t ledc_set_freq` (*ledc\_mode\_t speed\_mode, ledc\_timer\_t timer\_num, uint32\_t freq\_hz*)  
LEDC set channel frequency (Hz)

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_FAIL Can not find a proper pre-divider number base on the given frequency and the current `duty_resolution`.

#### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_num`: LEDC timer index (0-3), select from `ledc_timer_t`
- `freq_hz`: Set the LEDC frequency

`uint32_t ledc_get_freq` (*ledc\_mode\_t speed\_mode, ledc\_timer\_t timer\_num*)  
LEDC get channel frequency (Hz)

#### Return

- 0 error
- Others Current LEDC frequency

#### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_num`: LEDC timer index (0-3), select from `ledc_timer_t`

`esp_err_t ledc_set_duty` (*ledc\_mode\_t speed\_mode, ledc\_channel\_t channel, uint32\_t duty*)  
LEDC set duty Only after calling `ledc_update_duty` will the duty update.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `duty`: Set the LEDC duty, the range of duty setting is  $[0, (2^{**}duty\_resolution) - 1]$

`uint32_t ledc_get_duty` (*ledc\_mode\_t speed\_mode, ledc\_channel\_t channel*)  
LEDC get duty.

### Return

- LEDC\_ERR\_DUTY if parameter error
- Others Current LEDC duty

### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`

`esp_err_t ledc_set_fade` (*ledc\_mode\_t speed\_mode, ledc\_channel\_t channel, uint32\_t duty, ledc\_duty\_direction\_t gradule\_direction, uint32\_t step\_num, uint32\_t duty\_cyle\_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.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel (0-7), select from `ledc_channel_t`
- `duty`: Set the start of the gradient duty, the range of duty setting is  $[0, (2^{**}duty\_resolution) - 1]$
- `gradule_direction`: Set the direction of the gradient
- `step_num`: Set the number of the gradient
- `duty_cyle_num`: Set how many LEDC tick each time the gradient lasts
- `duty_scale`: Set gradient change amplitude

`esp_err_t ledc_isr_register` (*void (\*fn)*) *void \**, *void \*arg*, *int intr\_alloc\_flags*, *ledc\_isr\_handle\_t \*handle* Register LEDC interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Function pointer error.

### Parameters

- `fn`: Interrupt handler function.
- `arg`: User-supplied argument passed to the handler function.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `arg`: Parameter for handler function
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

`esp_err_t ledc_timer_set` (*ledc\_mode\_t speed\_mode, ledc\_timer\_t timer\_sel, uint32\_t clock\_divider, uint32\_t duty\_resolution, ledc\_clk\_src\_t clk\_src*)

Configure LEDC settings.

### Return

- (-1) Parameter error
- Other Current LEDC duty

### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: Timer index (0-3), there are 4 timers in LEDC module
- `clock_divider`: Timer clock divide value, the timer clock is divided from the selected clock source
- `duty_resolution`: Resolution of duty setting in number of bits. The range of duty values is  $[0, (2^{**}duty\_resolution) - 1]$
- `clk_src`: Select LEDC source clock.

`esp_err_t ledc_timer_rst` (*ledc\_mode\_t speed\_mode*, *uint32\_t timer\_sel*)  
Reset LEDC timer.

### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

`esp_err_t ledc_timer_pause` (*ledc\_mode\_t speed\_mode*, *uint32\_t timer\_sel*)  
Pause LEDC timer counter.

### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

`esp_err_t ledc_timer_resume` (*ledc\_mode\_t speed\_mode*, *uint32\_t timer\_sel*)  
Resume LEDC timer.

### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `timer_sel`: LEDC timer index (0-3), select from `ledc_timer_t`

`esp_err_t ledc_bind_channel_timer` (*ledc\_mode\_t speed\_mode*, `uint32_t channel`, `uint32_t timer_idx`)  
Bind LEDC channel with the selected timer.

**Return**

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

**Parameters**

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel index (0-7), select from `ledc_channel_t`
- `timer_idx`: LEDC timer index (0-3), select from `ledc_timer_t`

`esp_err_t ledc_set_fade_with_step` (*ledc\_mode\_t speed\_mode*, *ledc\_channel\_t channel*, `uint32_t target_duty`, `int scale`, `int cycle_num`)  
Set LEDC fade function. Should call `ledc_fade_func_install()` before calling this function. Call `ledc_fade_start()` after this to start fading.

**Return**

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` Fade function not installed.
- `ESP_FAIL` Fade function init error

**Parameters**

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from `ledc_channel_t`
- `target_duty`: Target duty of fading [ $0, (2^{**}duty\_resolution) - 1$ ]
- `scale`: Controls the increase or decrease step scale.
- `cycle_num`: increase or decrease the duty every `cycle_num` cycles

`esp_err_t ledc_set_fade_with_time` (*ledc\_mode\_t speed\_mode*, *ledc\_channel\_t channel*, `uint32_t target_duty`, `int max_fade_time_ms`)  
Set LEDC fade function, with a limited time. Should call `ledc_fade_func_install()` before calling this function. Call `ledc_fade_start()` after this to start fading.

**Return**

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` Fade function not installed.
- `ESP_FAIL` Fade function init error

**Parameters**

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode,
- `channel`: LEDC channel index (0-7), select from `ledc_channel_t`
- `target_duty`: Target duty of fading. ( $0 - (2^{**}duty\_resolution - 1)$ )

- `max_fade_time_ms`: The maximum time of the fading ( ms ).

`esp_err_t ledc_fade_func_install (int intr_alloc_flags)`

Install ledc fade function. This function will occupy interrupt of LEDC module.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` Fade function already installed.

#### Parameters

- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.

`void ledc_fade_func_uninstall ()`

Uninstall LEDC fade function.

`esp_err_t ledc_fade_start (ledc_mode_t speed_mode, ledc_channel_t channel, ledc_fade_mode_t wait_done)`

Start LEDC fading.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_STATE` Fade function not installed.
- `ESP_ERR_INVALID_ARG` Parameter error.

#### Parameters

- `speed_mode`: Select the LEDC `speed_mode`, high-speed mode and low-speed mode
- `channel`: LEDC channel number
- `wait_done`: Whether to block until fading done.

## Structures

`struct ledc_channel_config_t`

Configuration parameters of LEDC channel for `ledc_channel_config` function.

#### Public Members

`int gpio_num`

the LEDC output `gpio_num`, if you want to use `gpio16`, `gpio_num = 16`

`ledc_mode_t speed_mode`

LEDC speed `speed_mode`, high-speed mode or low-speed mode

`ledc_channel_t channel`

LEDC channel (0 - 7)

`ledc_intr_type_t intr_type`

configure interrupt, Fade interrupt enable or Fade interrupt disable

`ledc_timer_t timer_sel`

Select the timer source of channel (0 - 3)

`uint32_t duty`

LEDC channel duty, the range of duty setting is  $[0, (2^{**duty\_resolution}) - 1]$

`struct ledc_timer_config_t`

Configuration parameters of LEDC Timer timer for `ledc_timer_config` function.

## Public Members

`ledc_mode_t speed_mode`

LEDC speed `speed_mode`, high-speed mode or low-speed mode

`ledc_timer_bit_t duty_resolution`

LEDC channel duty resolution

`ledc_timer_bit_t bit_num`

Deprecated in ESP-IDF 3.0. This is an alias to ‘`duty_resolution`’ for backward compatibility with ESP-IDF 2.1

`ledc_timer_t timer_num`

The timer source of channel (0 - 3)

`uint32_t freq_hz`

LEDC timer frequency (Hz)

## Macros

`LEDC_APB_CLK_HZ`

`LEDC_REF_CLK_HZ`

`LEDC_ERR_DUTY`

## Type Definitions

`typedef intr_handle_t ledc_isr_handle_t`

## Enumerations

`enum ledc_mode_t`

*Values:*

`LEDC_HIGH_SPEED_MODE = 0`

LEDC high speed `speed_mode`

`LEDC_LOW_SPEED_MODE`

LEDC low speed `speed_mode`

`LEDC_SPEED_MODE_MAX`

LEDC speed limit

`enum ledc_intr_type_t`

*Values:*

`LEDC_INTR_DISABLE = 0`

Disable LEDC interrupt

**LEDC\_INTR\_FADE\_END**  
Enable LEDC interrupt

**enum ledc\_duty\_direction\_t**

*Values:*

**LEDC\_DUTY\_DIR\_DECREASE = 0**  
LEDC duty decrease direction

**LEDC\_DUTY\_DIR\_INCREASE = 1**  
LEDC duty increase direction

**enum ledc\_clk\_src\_t**

*Values:*

**LEDC\_REF\_TICK = 0**  
LEDC timer clock divided from reference tick (1Mhz)

**LEDC\_APB\_CLK**  
LEDC timer clock divided from APB clock (80Mhz)

**enum ledc\_timer\_t**

*Values:*

**LEDC\_TIMER\_0 = 0**  
LEDC timer 0

**LEDC\_TIMER\_1**  
LEDC timer 1

**LEDC\_TIMER\_2**  
LEDC timer 2

**LEDC\_TIMER\_3**  
LEDC timer 3

**enum ledc\_channel\_t**

*Values:*

**LEDC\_CHANNEL\_0 = 0**  
LEDC channel 0

**LEDC\_CHANNEL\_1**  
LEDC channel 1

**LEDC\_CHANNEL\_2**  
LEDC channel 2

**LEDC\_CHANNEL\_3**  
LEDC channel 3

**LEDC\_CHANNEL\_4**  
LEDC channel 4

**LEDC\_CHANNEL\_5**  
LEDC channel 5

**LEDC\_CHANNEL\_6**  
LEDC channel 6

**LEDC\_CHANNEL\_7**  
LEDC channel 7

**LEDC\_CHANNEL\_MAX**

**enum ledc\_timer\_bit\_t**

*Values:*

**LEDC\_TIMER\_10\_BIT** = 10  
LEDC PWM duty resolution of 10 bits

**LEDC\_TIMER\_11\_BIT** = 11  
LEDC PWM duty resolution of 11 bits

**LEDC\_TIMER\_12\_BIT** = 12  
LEDC PWM duty resolution of 12 bits

**LEDC\_TIMER\_13\_BIT** = 13  
LEDC PWM duty resolution of 13 bits

**LEDC\_TIMER\_14\_BIT** = 14  
LEDC PWM duty resolution of 14 bits

**LEDC\_TIMER\_15\_BIT** = 15  
LEDC PWM duty resolution of 15 bits

**enum ledc\_fade\_mode\_t**

*Values:*

**LEDC\_FADE\_NO\_WAIT** = 0  
LEDC fade function will return immediately

**LEDC\_FADE\_WAIT\_DONE**  
LEDC fade function will block until fading to the target duty

**LEDC\_FADE\_MAX**

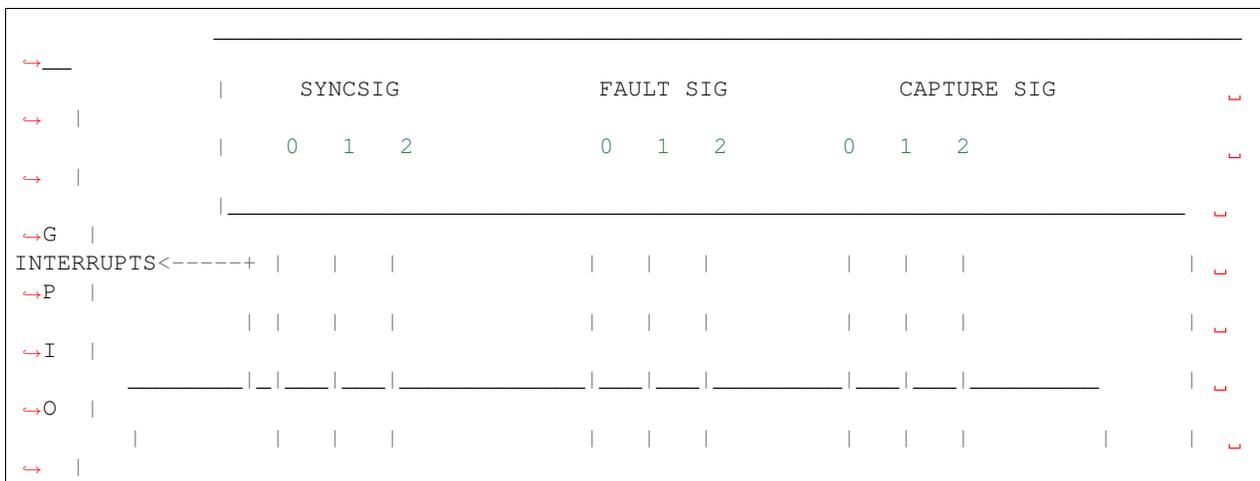
## 2.4.7 MCPWM

### Overview

ESP32 has two MCPWM units which can be used to control different motors.

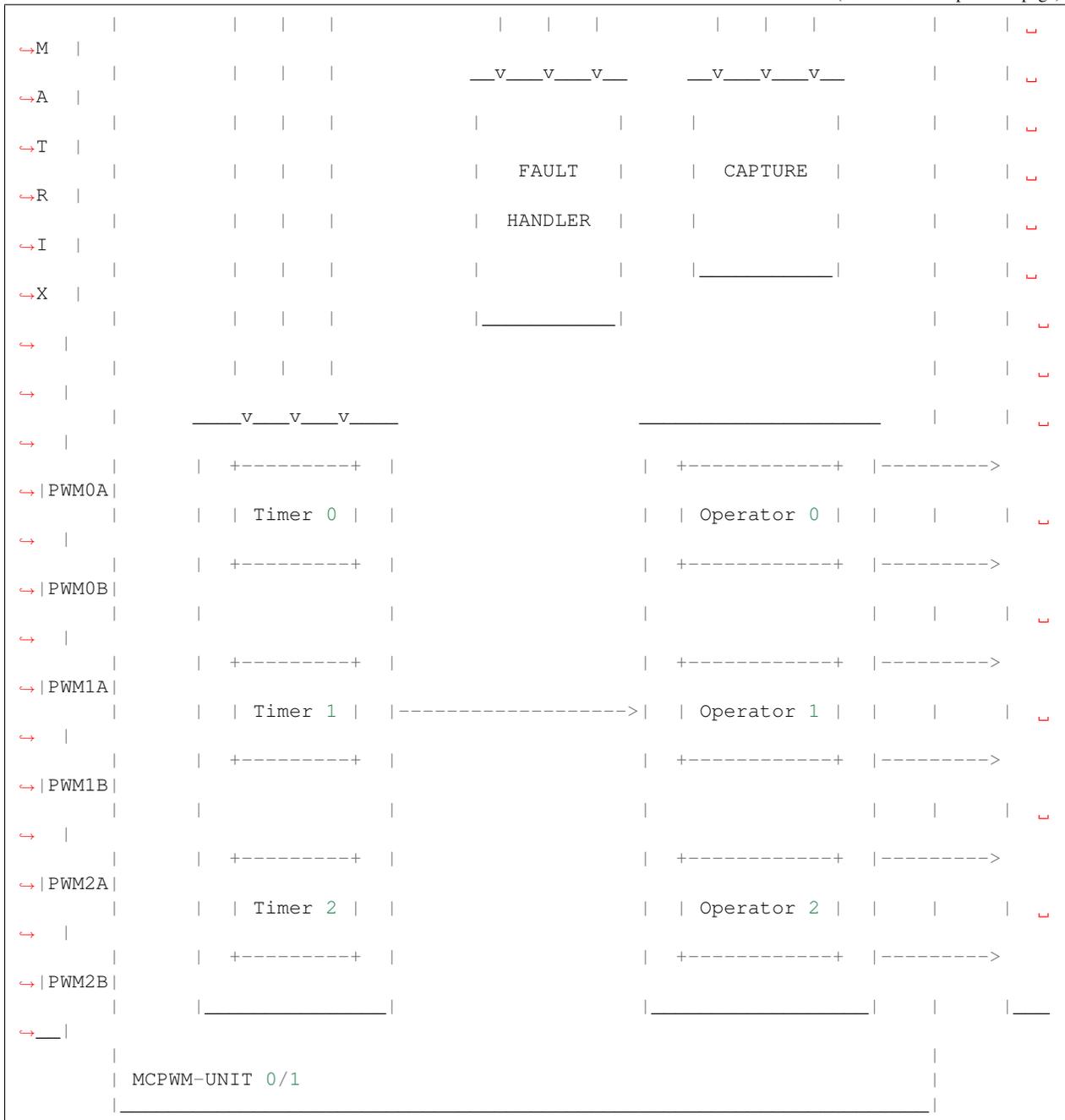
### Block Diagram

The block diagram of MCPWM unit is as shown.



(continues on next page)

(continued from previous page)



## Application Example

Examples of using MCPWM for motor control: [peripherals/mcpwm](#).

## API Reference

## Header File

- `driver/include/driver/mcpwm.h`

## Functions

`esp_err_t mcpwm_gpio_init` (*mcpwm\_unit\_t* *mcpwm\_num*, *mcpwm\_io\_signals\_t* *io\_signal*, int *gpio\_num*)  
This function initializes each gpio signal for MCPWM.

**Note** This function initializes one gpio at a time.

### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

### Parameters

- *mcpwm\_num*: set MCPWM Channel(0-1)
- *io\_signal*: set MCPWM signals, each MCPWM unit has 6 output(MCPWMXA, MCPWMXB) and 9 input(SYNC\_X, FAULT\_X, CAP\_X) 'X' is timer\_num(0-2)
- *gpio\_num*: set this to configure gpio for MCPWM, if you want to use gpio16, *gpio\_num* = 16

`esp_err_t mcpwm_set_pin` (*mcpwm\_unit\_t* *mcpwm\_num*, **const** *mcpwm\_pin\_config\_t* \**mcpwm\_pin*)  
Initialize MCPWM gpio structure.

**Note** This function can be used to initialize more then one gpio at a time.

### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

### Parameters

- *mcpwm\_num*: set MCPWM Channel(0-1)
- *mcpwm\_pin*: MCPWM pin structure

`esp_err_t mcpwm_init` (*mcpwm\_unit\_t* *mcpwm\_num*, *mcpwm\_timer\_t* *timer\_num*, **const** *mcpwm\_config\_t* \**mcpwm\_conf*)  
Initialize MCPWM parameters.

### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

### Parameters

- *mcpwm\_num*: set MCPWM Channel(0-1)
- *timer\_num*: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- *mcpwm\_conf*: configure structure *mcpwm\_config\_t*

`esp_err_t mcpwm_set_frequency` (*mcpwm\_unit\_t* mcpwm\_num, *mcpwm\_timer\_t* timer\_num, *uint32\_t* frequency)

Set frequency(in Hz) of MCPWM timer.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- frequency: set the frequency in Hz of each timer

`esp_err_t mcpwm_set_duty` (*mcpwm\_unit\_t* mcpwm\_num, *mcpwm\_timer\_t* timer\_num, *mcpwm\_operator\_t* op\_num, *float* duty)

Set duty cycle of each operator(MCPWMXA/MCPWMB)

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- op\_num: set the operator(MCPWMXA/MCPWMB), 'X' is timer number selected
- duty: set duty cycle in % (i.e for 62.3% duty cycle, duty = 62.3) of each operator

`esp_err_t mcpwm_set_duty_in_us` (*mcpwm\_unit\_t* mcpwm\_num, *mcpwm\_timer\_t* timer\_num, *mcpwm\_operator\_t* op\_num, *uint32\_t* duty)

Set duty cycle of each operator(MCPWMXA/MCPWMB) in us.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- op\_num: set the operator(MCPWMXA/MCPWMB), 'x' is timer number selected
- duty: set duty value in microseconds of each operator

`esp_err_t mcpwm_set_duty_type` (*mcpwm\_unit\_t* mcpwm\_num, *mcpwm\_timer\_t* timer\_num, *mcpwm\_operator\_t* op\_num, *mcpwm\_duty\_type\_t* duty\_num)

Set duty either active high or active low(out of phase/inverted)

**Note** Call this function every time after `mcpwm_set_signal_high` or `mcpwm_set_signal_low` to resume with previously set duty cycle

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `op_num`: set the operator(MCPWMXA/MCPWMXB), 'x' is timer number selected
- `duty_num`: set active low or active high duty type

`uint32_t mcpwm_get_frequency (mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num)`  
Get frequency of timer.

### Return

- frequency of timer

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`float mcpwm_get_duty (mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num, mcpwm_operator_t op_num)`  
Get duty cycle of each operator.

### Return

- duty cycle in % of each operator(56.7 means duty is 56.7%)

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `op_num`: set the operator(MCPWMXA/MCPWMXB), 'x' is timer number selected

`esp_err_t mcpwm_set_signal_high (mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num, mcpwm_operator_t op_num)`  
Use this function to set MCPWM signal high.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `op_num`: set the operator(MCPWMXA/MCPWMXB), 'x' is timer number selected

`esp_err_t mcpwm_set_signal_low(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num, mcpwm_operator_t op_num)`

Use this function to set MCPWM signal low.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- op\_num: set the operator(MCPWMXA/MCPWMB), 'x' is timer number selected

`esp_err_t mcpwm_start(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num)`

Start MCPWM signal on timer 'x'.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_stop(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num)`

Start MCPWM signal on timer 'x'.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_carrier_init(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num, const mcpwm_carrier_config_t *carrier_conf)`

Initialize carrier configuration.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- mcpwm\_num: set MCPWM unit(0-1)
- timer\_num: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- carrier\_conf: configure structure `mcpwm_carrier_config_t`

`esp_err_t mcpwm_carrier_enable(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num)`  
Enable MCPWM carrier submodule, for respective timer.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_carrier_disable(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num)`  
Disable MCPWM carrier submodule, for respective timer.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_carrier_set_period(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num, uint8_t carrier_period)`  
Set period of carrier.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `carrier_period`: set the carrier period of each timer, carrier period = (carrier\_period + 1)\*800ns (carrier\_period <= 15)

`esp_err_t mcpwm_carrier_set_duty_cycle(mcpwm_unit_t mcpwm_num, mcpwm_timer_t timer_num, uint8_t carrier_duty)`  
Set duty\_cycle of carrier.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

- `carrier_duty`: set `duty_cycle` of carrier , carrier duty cycle = `carrier_duty*12.5%` (`chop_duty <= 7`)

`esp_err_t mcpwm_carrier_oneshot_mode_enable` (*mcpwm\_unit\_t* *mcpwm\_num*, *mcpwm\_timer\_t* *timer\_num*, *uint8\_t* *pulse\_width*)

Enable and set width of first pulse in carrier oneshot mode.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `pulse_width`: set pulse width of first pulse in oneshot mode, width = (carrier period)\*(pulse\_width +1) (pulse\_width <= 15)

`esp_err_t mcpwm_carrier_oneshot_mode_disable` (*mcpwm\_unit\_t* *mcpwm\_num*, *mcpwm\_timer\_t* *timer\_num*)

Disable oneshot mode, width of first pulse = carrier period.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_carrier_output_invert` (*mcpwm\_unit\_t* *mcpwm\_num*, *mcpwm\_timer\_t* *timer\_num*, *mcpwm\_carrier\_out\_ivt\_t* *carrier\_ivt\_mode*)

Enable or disable carrier output inversion.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `carrier_ivt_mode`: enable or disable carrier output inversion

`esp_err_t mcpwm_deadtime_enable` (*mcpwm\_unit\_t* *mcpwm\_num*, *mcpwm\_timer\_t* *timer\_num*, *mcpwm\_deadtime\_type\_t* *dt\_mode*, *uint32\_t* *red*, *uint32\_t* *fed*)

Enable and initialize deadtime for each MCPWM timer.

#### Return

- `ESP_OK` Success

- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `dt_mode`: set deadtime mode
- `red`: set rising edge delay = `red*100ns`
- `fed`: set rising edge delay = `fed*100ns`

`esp_err_t mcpwm_deadtime_disable` (*mcpwm\_unit\_t* `mcpwm_num`, *mcpwm\_timer\_t* `timer_num`)  
Disable deadtime on MCPWM timer.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_fault_init` (*mcpwm\_unit\_t* `mcpwm_num`, *mcpwm\_fault\_input\_level\_t* `input_level`,  
*mcpwm\_fault\_signal\_t* `fault_sig`)  
Initialize fault submodule, currently low level triggering not supported.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `input_level`: set fault signal level, which will cause fault to occur
- `fault_sig`: set the fault Pin, which needs to be enabled

`esp_err_t mcpwm_fault_set_oneshot_mode` (*mcpwm\_unit\_t* `mcpwm_num`, *mcpwm\_timer\_t* `timer_num`,  
*mcpwm\_fault\_signal\_t* `fault_sig`, *mcpwm\_action\_on\_pwmxa\_t* `action_on_pwmxa`,  
*mcpwm\_action\_on\_pwmxb\_t* `action_on_pwmxb`)

Set oneshot mode on fault detection, once fault occur in oneshot mode reset is required to resume MCPWM signals.

**Note** currently low level triggering not supported

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)

- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `fault_sig`: set the fault Pin, which needs to be enabled for oneshot mode
- `action_on_pwmxa`: action to be taken on MCPWMXA when fault occurs, either no change or high or low or toggle
- `action_on_pwmxb`: action to be taken on MCPWMXB when fault occurs, either no change or high or low or toggle

`esp_err_t mcpwm_fault_set_cyc_mode` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_timer\_t timer\_num, mcpwm\_fault\_signal\_t fault\_sig, mcpwm\_action\_on\_pwmxa\_t action\_on\_pwmxa, mcpwm\_action\_on\_pwmxb\_t action\_on\_pwmxb*)

Set cycle-by-cycle mode on fault detection, once fault occur in cyc mode MCPWM signal resumes as soon as fault signal becomes inactive.

**Note** currently low level triggering not supported

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `fault_sig`: set the fault Pin, which needs to be enabled for cyc mode
- `action_on_pwmxa`: action to be taken on MCPWMXA when fault occurs, either no change or high or low or toggle
- `action_on_pwmxb`: action to be taken on MCPWMXB when fault occurs, either no change or high or low or toggle

`esp_err_t mcpwm_fault_deinit` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_fault\_signal\_t fault\_sig*)

Disable fault signal.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `fault_sig`: fault pin, which needs to be disabled

`esp_err_t mcpwm_capture_enable` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_capture\_signal\_t cap\_sig, mcpwm\_capture\_on\_edge\_t cap\_edge, uint32\_t num\_of\_pulse*)

Initialize capture submodule.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_edge`: set capture edge, BIT(0) - negative edge, BIT(1) - positive edge
- `cap_sig`: capture Pin, which needs to be enabled
- `num_of_pulse`: count time between rising/falling edge between 2 \*(pulses mentioned), counter uses APB\_CLK

`esp_err_t mcpwm_capture_disable` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_capture\_signal\_t cap\_sig*)  
Disable capture signal.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_sig`: capture Pin, which needs to be disabled

`uint32_t mcpwm_capture_signal_get_value` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_capture\_signal\_t cap\_sig*)

Get capture value.

**Return** Captured value

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `cap_sig`: capture pin on which value is to be measured

`uint32_t mcpwm_capture_signal_get_edge` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_capture\_signal\_t cap\_sig*)

Get edge of capture signal.

**Return** Capture signal edge: 1 - positive edge, 2 - negative edge

### Parameters

- `mcpwm_num`: set MCPWM Channel(0-1)
- `cap_sig`: capture pin of whose edge is to be determined

`esp_err_t mcpwm_sync_enable` (*mcpwm\_unit\_t mcpwm\_num, mcpwm\_timer\_t timer\_num, mcpwm\_sync\_signal\_t sync\_sig, uint32\_t phase\_val*)

Initialize sync submodule.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)

- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers
- `sync_sig`: set the fault Pin, which needs to be enabled
- `phase_val`: phase value in 1/1000(for 86.7%, `phase_val = 867`) which timer moves to on sync signal

`esp_err_t mcpwm_sync_disable` (*mcpwm\_unit\_t* `mcpwm_num`, *mcpwm\_timer\_t* `timer_num`)  
Disable sync submodule on given timer.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `timer_num`: set timer number(0-2) of MCPWM, each MCPWM unit has 3 timers

`esp_err_t mcpwm_isr_register` (*mcpwm\_unit\_t* `mcpwm_num`, void (\**fn*)) void \*  
, void \**arg*, int *intr\_alloc\_flags*, *intr\_handle\_t* \**handle* Register MCPWM interrupt handler, the handler is an ISR. the handler will be attached to the same CPU core that this function is running on.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Function pointer error.

#### Parameters

- `mcpwm_num`: set MCPWM unit(0-1)
- `fn`: interrupt handler function.
- `arg`: user-supplied argument passed to the handler function.
- `intr_alloc_flags`: flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. see `esp_intr_alloc.h` for more info.
- `arg`: parameter for handler function
- `handle`: pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

## Structures

`struct mcpwm_pin_config_t`  
MCPWM pin number for.

#### Public Members

int `mcpwm0a_out_num`  
MCPWM0A out pin

int `mcpwm0b_out_num`  
MCPWM0A out pin

```
int mcpwm1a_out_num  
    MCPWM0A out pin  
  
int mcpwm1b_out_num  
    MCPWM0A out pin  
  
int mcpwm2a_out_num  
    MCPWM0A out pin  
  
int mcpwm2b_out_num  
    MCPWM0A out pin  
  
int mcpwm_sync0_in_num  
    SYNC0 in pin  
  
int mcpwm_sync1_in_num  
    SYNC1 in pin  
  
int mcpwm_sync2_in_num  
    SYNC2 in pin  
  
int mcpwm_fault0_in_num  
    FAULT0 in pin  
  
int mcpwm_fault1_in_num  
    FAULT1 in pin  
  
int mcpwm_fault2_in_num  
    FAULT2 in pin  
  
int mcpwm_cap0_in_num  
    CAP0 in pin  
  
int mcpwm_cap1_in_num  
    CAP1 in pin  
  
int mcpwm_cap2_in_num  
    CAP2 in pin
```

```
struct mcpwm_config_t  
    MCPWM config structure.
```

### Public Members

```
uint32_t frequency  
    Set frequency of MCPWM in Hz  
  
float cmp_r_a  
    Set % duty cycle for operator a(MCPWMXA), i.e for 62.3% duty cycle, duty_a = 62.3  
  
float cmp_r_b  
    Set % duty cycle for operator b(MCPWMXB), i.e for 48% duty cycle, duty_b = 48.0  
  
mcpwm_duty_type_t duty_mode  
    Set type of duty cycle  
  
mcpwm_counter_type_t counter_mode  
    Set type of MCPWM counter
```

```
struct mcpwm_carrier_config_t  
    MCPWM config carrier structure.
```

## Public Members

`uint8_t carrier_period`

Set carrier period = (carrier\_period + 1)\*800ns, carrier\_period should be < 16

`uint8_t carrier_duty`

Set carrier duty cycle, carrier\_duty should be less then 8(increment every 12.5%)

`uint8_t pulse_width_in_os`

Set pulse width of first pulse in one shot mode = (carrier period)\*(pulse\_width\_in\_os + 1), should be less then 16

`mcpwm_carrier_os_t carrier_os_mode`

Enable or disable carrier oneshot mode

`mcpwm_carrier_out_ivt_t carrier_ivt_mode`

Invert output of carrier

## Enumerations

`enum mcpwm_io_signals_t`

IO signals for MCPWM 6 MCPWM output pins that generate PWM signals 3 MCPWM fault input pins to detect faults like overcurrent, overvoltage, etc 3 MCPWM sync input pins to synchronize MCPWM outputs signals 3 MCPWM capture input pin to capture hall sell signal to measure time.

*Values:*

**MCPWM0A** = 0

PWM0A output pin

**MCPWM0B**

PWM0B output pin

**MCPWM1A**

PWM1A output pin

**MCPWM1B**

PWM1B output pin

**MCPWM2A**

PWM2A output pin

**MCPWM2B**

PWM2B output pin

**MCPWM\_SYNC\_0**

SYNC0 input pin

**MCPWM\_SYNC\_1**

SYNC1 input pin

**MCPWM\_SYNC\_2**

SYNC2 input pin

**MCPWM\_FAULT\_0**

FAULT0 input pin

**MCPWM\_FAULT\_1**

FAULT1 input pin

**MCPWM\_FAULT\_2**  
FAULT2 input pin

**MCPWM\_CAP\_0 = 84**  
CAP0 input pin

**MCPWM\_CAP\_1**  
CAP1 input pin

**MCPWM\_CAP\_2**  
CAP2 input pin

**enum mcpwm\_unit\_t**  
Select MCPWM unit.

*Values:*

**MCPWM\_UNIT\_0 = 0**  
MCPWM unit0 selected

**MCPWM\_UNIT\_1**  
MCPWM unit1 selected

**MCPWM\_UNIT\_MAX**  
Num of MCPWM units on ESP32

**enum mcpwm\_timer\_t**  
Select MCPWM timer.

*Values:*

**MCPWM\_TIMER\_0 = 0**  
Select MCPWM timer0

**MCPWM\_TIMER\_1**  
Select MCPWM timer1

**MCPWM\_TIMER\_2**  
Select MCPWM timer2

**MCPWM\_TIMER\_MAX**  
Num of MCPWM timers on ESP32

**enum mcpwm\_operator\_t**  
Select MCPWM operator.

*Values:*

**MCPWM\_OPR\_A = 0**  
Select MCPWMXA, where 'X' is timer number

**MCPWM\_OPR\_B**  
Select MCPWMXB, where 'X' is timer number

**MCPWM\_OPR\_MAX**  
Num of operators to each timer of MCPWM

**enum mcpwm\_counter\_type\_t**  
Select type of MCPWM counter.

*Values:*

**MCPWM\_UP\_COUNTER = 1**  
For asymmetric MCPWM

**MCPWM\_DOWN\_COUNTER**

For asymmetric MCPWM

**MCPWM\_UP\_DOWN\_COUNTER**

For symmetric MCPWM, frequency is half of MCPWM frequency set

**MCPWM\_COUNTER\_MAX**

Maximum counter mode

**enum mcpwm\_duty\_type\_t**

Select type of MCPWM duty cycle mode.

*Values:***MCPWM\_DUTY\_MODE\_0 = 0**

Active high duty, i.e. duty cycle proportional to high time for asymmetric MCPWM

**MCPWM\_DUTY\_MODE\_1**

Active low duty, i.e. duty cycle proportional to low time for asymmetric MCPWM, out of phase(inverted) MCPWM

**MCPWM\_DUTY\_MODE\_MAX**

Num of duty cycle modes

**enum mcpwm\_carrier\_os\_t**

MCPWM carrier oneshot mode, in this mode the width of the first pulse of carrier can be programmed.

*Values:***MCPWM\_ONESHOT\_MODE\_DIS = 0**

Enable oneshot mode

**MCPWM\_ONESHOT\_MODE\_EN**

Disable oneshot mode

**enum mcpwm\_carrier\_out\_ivt\_t**

MCPWM carrier output inversion, high frequency carrier signal active with MCPWM signal is high.

*Values:***MCPWM\_CARRIER\_OUT\_IVT\_DIS = 0**

Enable carrier output inversion

**MCPWM\_CARRIER\_OUT\_IVT\_EN**

Disable carrier output inversion

**enum mcpwm\_sync\_signal\_t**

MCPWM select sync signal input.

*Values:***MCPWM\_SELECT\_SYNC0 = 4**

Select SYNC0 as input

**MCPWM\_SELECT\_SYNC1**

Select SYNC1 as input

**MCPWM\_SELECT\_SYNC2**

Select SYNC2 as input

**enum mcpwm\_fault\_signal\_t**

MCPWM select fault signal input.

*Values:*

**MCPWM\_SELECT\_F0** = 0

Select F0 as input

**MCPWM\_SELECT\_F1**

Select F1 as input

**MCPWM\_SELECT\_F2**

Select F2 as input

**enum mcpwm\_fault\_input\_level\_t**

MCPWM select triggering level of fault signal.

*Values:*

**MCPWM\_LOW\_LEVEL\_TGR** = 0

Fault condition occurs when fault input signal goes from high to low, currently not supported

**MCPWM\_HIGH\_LEVEL\_TGR**

Fault condition occurs when fault input signal goes low to high

**enum mcpwm\_action\_on\_pwmxa\_t**

MCPWM select action to be taken on MCPWMXA when fault occurs.

*Values:*

**MCPWM\_NO\_CHANGE\_IN\_MCPWMXA** = 0

No change in MCPWMXA output

**MCPWM\_FORCE\_MCPWMXA\_LOW**

Make MCPWMXA output low

**MCPWM\_FORCE\_MCPWMXA\_HIGH**

Make MCPWMXA output high

**MCPWM\_TOG\_MCPWMXA**

Make MCPWMXA output toggle

**enum mcpwm\_action\_on\_pwmxb\_t**

MCPWM select action to be taken on MCPWMxB when fault occurs.

*Values:*

**MCPWM\_NO\_CHANGE\_IN\_MCPWMXB** = 0

No change in MCPWMXB output

**MCPWM\_FORCE\_MCPWMXB\_LOW**

Make MCPWMXB output low

**MCPWM\_FORCE\_MCPWMXB\_HIGH**

Make MCPWMXB output high

**MCPWM\_TOG\_MCPWMXB**

Make MCPWMXB output toggle

**enum mcpwm\_capture\_signal\_t**

MCPWM select capture signal input.

*Values:*

**MCPWM\_SELECT\_CAP0** = 0

Select CAP0 as input

**MCPWM\_SELECT\_CAP1**

Select CAP1 as input

**MCPWM\_SELECT\_CAP2**

Select CAP2 as input

**enum mcpwm\_capture\_on\_edge\_t**

MCPWM select capture starts from which edge.

*Values:***MCPWM\_NEG\_EDGE = 0**

Capture starts from negative edge

**MCPWM\_POS\_EDGE**

Capture starts from positive edge

**enum mcpwm\_deadtime\_type\_t**

MCPWM deadtime types, used to generate deadtime, RED refers to rising edge delay and FED refers to falling edge delay.

*Values:***MCPWM\_BYPASS\_RED = 0**

MCPWMXA = no change, MCPWMXB = falling edge delay

**MCPWM\_BYPASS\_FED**

MCPWMXA = rising edge delay, MCPWMXB = no change

**MCPWM\_ACTIVE\_HIGH\_MODE**

MCPWMXA = rising edge delay, MCPWMXB = falling edge delay

**MCPWM\_ACTIVE\_LOW\_MODE**

MCPWMXA = compliment of rising edge delay, MCPWMXB = compliment of falling edge delay

**MCPWM\_ACTIVE\_HIGH\_COMPLIMENT\_MODE**

MCPWMXA = rising edge delay, MCPWMXB = compliment of falling edge delay

**MCPWM\_ACTIVE\_LOW\_COMPLIMENT\_MODE**

MCPWMXA = compliment of rising edge delay, MCPWMXB = falling edge delay

**MCPWM\_ACTIVE\_RED\_FED\_FROM\_PWMXA**

MCPWMXA = MCPWMXB = rising edge delay as well as falling edge delay, generated from MCPWMXA

**MCPWM\_ACTIVE\_RED\_FED\_FROM\_PWMXB**

MCPWMXA = MCPWMXB = rising edge delay as well as falling edge delay, generated from MCPWMXB

**MCPWM\_DEADTIME\_TYPE\_MAX**

## 2.4.8 Pulse Counter

### Introduction

The PCNT (Pulse Counter) module is designed to count the number of rising and/or falling edges of an input signal. Each pulse counter unit has a 16-bit signed counter register and two channels that can be configured to either increment or decrement the counter. Each channel has a signal input that accepts signal edges to be detected, as well as a control input that can be used to enable or disable the signal input. The inputs have optional filters that can be used to discard unwanted glitches in the signal.

## Functionality Overview

Description of functionality of this API has been broken down into four sections:

- *Configuration* - describes counter's configuration parameters and how to setup the counter.
- *Operating the Counter* - provides information on control functions to pause, measure and clear the counter.
- *Filtering Pulses* - describes options to filtering pulses and the counter control signals.
- *Using Interrupts* - presents how to trigger interrupts on specific states of the counter.

## Configuration

The PCNT module has eight independent counting “units” numbered from 0 to 7. In the API they are referred to using `pcnt_unit_t`. Each unit has two independent channels numbered as 0 and 1 and specified with `pcnt_channel_t`.

The configuration is provided separately per unit's channel using `pcnt_config_t` and covers:

- The unit and the channel number this configuration refers to.
- GPIO numbers of the pulse input and the pulse gate input.
- Two pairs of parameters: `pcnt_ctrl_mode_t` and `pcnt_count_mode_t` to define how the counter reacts depending on the the status of control signal and how counting is done positive / negative edge of the pulses.
- Two limit values (minimum / maximum) that are used to establish watchpoints and trigger interrupts when the pulse count is meeting particular limit.

Setting up of particular channel is then done by calling a function `pcnt_unit_config()` with above `pcnt_config_t` as the input parameter.

To disable the pulse or the control input pin in configuration, provide `PCNT_PIN_NOT_USED` instead of the GPIO number.

## Operating the Counter

After doing setup with `pcnt_unit_config()`, the counter immediately starts to operate. The accumulated pulse count can be checked by calling `pcnt_get_counter_value()`.

There are couple of functions that allow to control the counter's operation: `pcnt_counter_pause()`, `pcnt_counter_resume()` and `pcnt_counter_clear()`

It is also possible to dynamically change the previously set up counter modes with `pcnt_unit_config()` by calling `pcnt_set_mode()`.

If desired, the pulse input pin and the control input pin may be changed “on the fly” using `pcnt_set_pin()`. To disable particular input provide as a function parameter `PCNT_PIN_NOT_USED` instead of the GPIO number.

---

**Note:** For the counter not to miss any pulses, the pulse duration should be longer than one `APB_CLK` cycle (12.5 ns). The pulses are sampled on the edges of the `APB_CLK` clock and may be missed, if fall between the edges. This applies to counter operation with or without a *filer*.

---

## Filtering Pulses

The PCNT unit features filters on each of the pulse and control inputs, adding the option to ignore short glitches in the signals.

The length of ignored pulses is provided in APB\_CLK clock cycles by calling `pcnt_set_filter_value()`. The current filter setting may be checked with `pcnt_get_filter_value()`. The APB\_CLK clock is running at 80 MHz.

The filter is put into operation / suspended by calling `pcnt_filter_enable()` / `pcnt_filter_disable()`.

## Using Interrupts

There are five counter state watch events, defined in `pcnt_evt_type_t`, that are able to trigger an interrupt. The event happens on the pulse counter reaching specific values:

- Minimum or maximum count values: `counter_l_lim` or `counter_h_lim` provided in `pcnt_config_t` as discussed in *Configuration*
- Threshold 0 or Threshold 1 values set using function `pcnt_set_event_value()`.
- Pulse count = 0

To register, enable or disable an interrupt to service the above events, call `pcnt_isr_register()`, `pcnt_intr_enable()`. and `pcnt_intr_disable()`. To enable or disable events on reaching threshold values, you will also need to call functions `pcnt_event_enable()` and `pcnt_event_disable()`.

In order to check what are the threshold values currently set, use function `pcnt_get_event_value()`.

## Application Example

Pulse counter with control signal and event interrupt example: [peripherals/pcnt](#).

## API Reference

### Header File

- `driver/include/driver/pcnt.h`

### Functions

`esp_err_t pcnt_unit_config(const pcnt_config_t *pcnt_config)`  
Configure Pulse Counter unit.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `pcnt_config`: Pointer of Pulse Counter unit configure parameter

`esp_err_t pcnt_get_counter_value(pcnt_unit_t pcnt_unit, int16_t *count)`  
Get pulse counter value.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `pcnt_unit`: Pulse Counter unit number
- `count`: Pointer to accept counter value

`esp_err_t pcnt_counter_pause` (*pcnt\_unit\_t pcnt\_unit*)  
Pause PCNT counter of PCNT unit.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `pcnt_unit`: PCNT unit number

`esp_err_t pcnt_counter_resume` (*pcnt\_unit\_t pcnt\_unit*)  
Resume counting for PCNT counter.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `pcnt_unit`: PCNT unit number, select from `pcnt_unit_t`

`esp_err_t pcnt_counter_clear` (*pcnt\_unit\_t pcnt\_unit*)  
Clear and reset PCNT counter value to zero.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `pcnt_unit`: PCNT unit number, select from `pcnt_unit_t`

`esp_err_t pcnt_intr_enable` (*pcnt\_unit\_t pcnt\_unit*)  
Enable PCNT interrupt for PCNT unit.

**Note** Each Pulse counter unit has five watch point events that share the same interrupt. Configure events with `pcnt_event_enable()` and `pcnt_event_disable()`

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `pcnt_unit`: PCNT unit number

`esp_err_t pcnt_intr_disable` (*pcnt\_unit\_t pcnt\_unit*)  
Disable PCNT interrupt for PCNT unit.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `pcnt_unit`: PCNT unit number

`esp_err_t pcnt_event_enable` (*pcnt\_unit\_t unit, pcnt\_evt\_type\_t evt\_type*)  
Enable PCNT event of PCNT unit.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `unit`: PCNT unit number
- `evt_type`: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).

`esp_err_t pcnt_event_disable` (*pcnt\_unit\_t unit, pcnt\_evt\_type\_t evt\_type*)  
Disable PCNT event of PCNT unit.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `unit`: PCNT unit number
- `evt_type`: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).

`esp_err_t pcnt_set_event_value` (*pcnt\_unit\_t unit, pcnt\_evt\_type\_t evt\_type, int16\_t value*)  
Set PCNT event value of PCNT unit.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `unit`: PCNT unit number
- `evt_type`: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).
- `value`: Counter value for PCNT event

`esp_err_t pcnt_get_event_value` (*pcnt\_unit\_t* unit, *pcnt\_evt\_type\_t* evt\_type, int16\_t \*value)  
Get PCNT event value of PCNT unit.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- unit: PCNT unit number
- evt\_type: Watch point event type. All enabled events share the same interrupt (one interrupt per pulse counter unit).
- value: Pointer to accept counter value for PCNT event

`esp_err_t pcnt_isr_register` (void (\*fn)) void \*  
, void \*arg, int intr\_alloc\_flags, *pcnt\_isr\_handle\_t* \*handle Register PCNT interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Function pointer error.

**Parameters**

- fn: Interrupt handler function.
- arg: Parameter for handler function
- intr\_alloc\_flags: Flags used to allocate the interrupt. One or multiple (ORred) ESP\_INTR\_FLAG\_\* values. See esp\_intr\_alloc.h for more info.
- handle: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

`esp_err_t pcnt_set_pin` (*pcnt\_unit\_t* unit, *pcnt\_channel\_t* channel, int pulse\_io, int ctrl\_io)  
Configure PCNT pulse signal input pin and control input pin.

**Note** Set the signal input to PCNT\_PIN\_NOT\_USED if unused.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- unit: PCNT unit number
- channel: PCNT channel number
- pulse\_io: Pulse signal input GPIO
- ctrl\_io: Control signal input GPIO

`esp_err_t pcnt_filter_enable` (*pcnt\_unit\_t* unit)  
Enable PCNT input filter.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `unit`: PCNT unit number

`esp_err_t pcnt_filter_disable` (*pcnt\_unit\_t unit*)  
Disable PCNT input filter.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `unit`: PCNT unit number

`esp_err_t pcnt_set_filter_value` (*pcnt\_unit\_t unit, uint16\_t filter\_val*)  
Set PCNT filter value.

**Note** `filter_val` is a 10-bit value, so the maximum `filter_val` should be limited to 1023.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `unit`: PCNT unit number
- `filter_val`: PCNT signal filter value, counter in APB\_CLK cycles. Any pulses lasting shorter than this will be ignored when the filter is enabled.

`esp_err_t pcnt_get_filter_value` (*pcnt\_unit\_t unit, uint16\_t \*filter\_val*)  
Get PCNT filter value.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `unit`: PCNT unit number
- `filter_val`: Pointer to accept PCNT filter value.

`esp_err_t pcnt_set_mode` (*pcnt\_unit\_t unit, pcnt\_channel\_t channel, pcnt\_count\_mode\_t pos\_mode, pcnt\_count\_mode\_t neg\_mode, pcnt\_ctrl\_mode\_t hctrl\_mode, pcnt\_ctrl\_mode\_t lctrl\_mode*)  
Set PCNT counter mode.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

### Parameters

- `unit`: PCNT unit number
- `channel`: PCNT channel number
- `pos_mode`: Counter mode when detecting positive edge
- `neg_mode`: Counter mode when detecting negative edge
- `hctrl_mode`: Counter mode when control signal is high level
- `lctrl_mode`: Counter mode when control signal is low level

### Structures

#### **struct pcnt\_config\_t**

Pulse Counter configuration for a single channel.

#### **Public Members**

##### **int pulse\_gpio\_num**

Pulse input GPIO number, if you want to use GPIO16, enter `pulse_gpio_num = 16`, a negative value will be ignored

##### **int ctrl\_gpio\_num**

Control signal input GPIO number, a negative value will be ignored

##### *pcnt\_ctrl\_mode\_t* **lctrl\_mode**

PCNT low control mode

##### *pcnt\_ctrl\_mode\_t* **hctrl\_mode**

PCNT high control mode

##### *pcnt\_count\_mode\_t* **pos\_mode**

PCNT positive edge count mode

##### *pcnt\_count\_mode\_t* **neg\_mode**

PCNT negative edge count mode

##### **int16\_t counter\_h\_lim**

Maximum counter value

##### **int16\_t counter\_l\_lim**

Minimum counter value

##### *pcnt\_unit\_t* **unit**

PCNT unit number

##### *pcnt\_channel\_t* **channel**

the PCNT channel

### Macros

#### **PCNT\_PIN\_NOT\_USED**

When selected for a pin, this pin will not be used

## Type Definitions

```
typedef intr_handle_t pcnt_isr_handle_t
```

## Enumerations

```
enum pcnt_ctrl_mode_t
```

Selection of available modes that determine the counter's action depending on the state of the control signal's input GPIO.

**Note** Configuration covers two actions, one for high, and one for low level on the control input

*Values:*

```
PCNT_MODE_KEEP = 0
```

Control mode: won't change counter mode

```
PCNT_MODE_REVERSE = 1
```

Control mode: invert counter mode(increase -> decrease, decrease -> increase)

```
PCNT_MODE_DISABLE = 2
```

Control mode: Inhibit counter(counter value will not change in this condition)

```
PCNT_MODE_MAX
```

```
enum pcnt_count_mode_t
```

Selection of available modes that determine the counter's action on the edge of the pulse signal's input GPIO.

**Note** Configuration covers two actions, one for positive, and one for negative edge on the pulse input

*Values:*

```
PCNT_COUNT_DIS = 0
```

Counter mode: Inhibit counter(counter value will not change in this condition)

```
PCNT_COUNT_INC = 1
```

Counter mode: Increase counter value

```
PCNT_COUNT_DEC = 2
```

Counter mode: Decrease counter value

```
PCNT_COUNT_MAX
```

```
enum pcnt_unit_t
```

Selection of all available PCNT units.

*Values:*

```
PCNT_UNIT_0 = 0
```

PCNT unit 0

```
PCNT_UNIT_1 = 1
```

PCNT unit 1

```
PCNT_UNIT_2 = 2
```

PCNT unit 2

```
PCNT_UNIT_3 = 3
```

PCNT unit 3

**PCNT\_UNIT\_4** = 4  
PCNT unit 4

**PCNT\_UNIT\_5** = 5  
PCNT unit 5

**PCNT\_UNIT\_6** = 6  
PCNT unit 6

**PCNT\_UNIT\_7** = 7  
PCNT unit 7

**PCNT\_UNIT\_MAX**

**enum pcnt\_channel\_t**

Selection of channels available for a single PCNT unit.

*Values:*

**PCNT\_CHANNEL\_0** = 0x00  
PCNT channel 0

**PCNT\_CHANNEL\_1** = 0x01  
PCNT channel 1

**PCNT\_CHANNEL\_MAX**

**enum pcnt\_evt\_type\_t**

Selection of counter's events the may trigger an interrupt.

*Values:*

**PCNT\_EVT\_L\_LIM** = 0  
PCNT watch point event: Minimum counter value

**PCNT\_EVT\_H\_LIM** = 1  
PCNT watch point event: Maximum counter value

**PCNT\_EVT\_THRES\_0** = 2  
PCNT watch point event: threshold0 value event

**PCNT\_EVT\_THRES\_1** = 3  
PCNT watch point event: threshold1 value event

**PCNT\_EVT\_ZERO** = 4  
PCNT watch point event: counter value zero event

**PCNT\_EVT\_MAX**

## 2.4.9 RMT

The RMT (Remote Control) module driver can be used to send and receive infrared remote control signals. Due to flexibility of RMT module, the driver can also be used to generate or receive many other types of signals.

The signal, which consists of a series of pulses, is generated by RMT's transmitter based on a list of values. The values define the pulse duration and a binary level, see below. The transmitter can also provide a carrier and modulate it with provided pulses.

The reverse operation is performed by the receiver, where a series of pulses is decoded into a list of values containing the pulse duration and binary level. A filter may be applied to remove high frequency noise from the input signal.

There couple of typical steps to setup and operate the RMT and they are discussed in the following sections:

1. *Configure Driver*

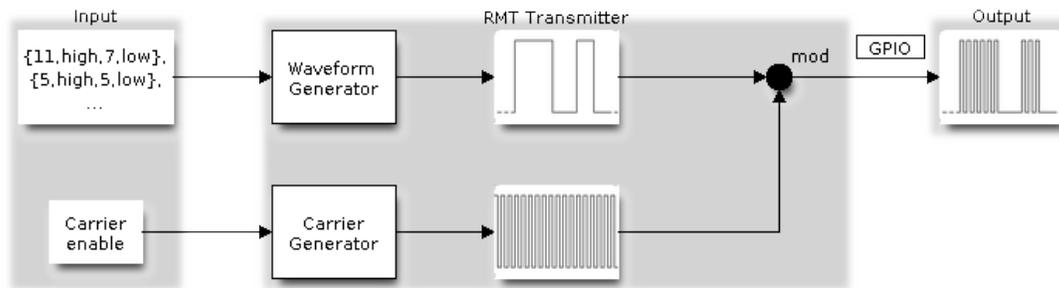


Fig. 6: RMT Transmitter Overview

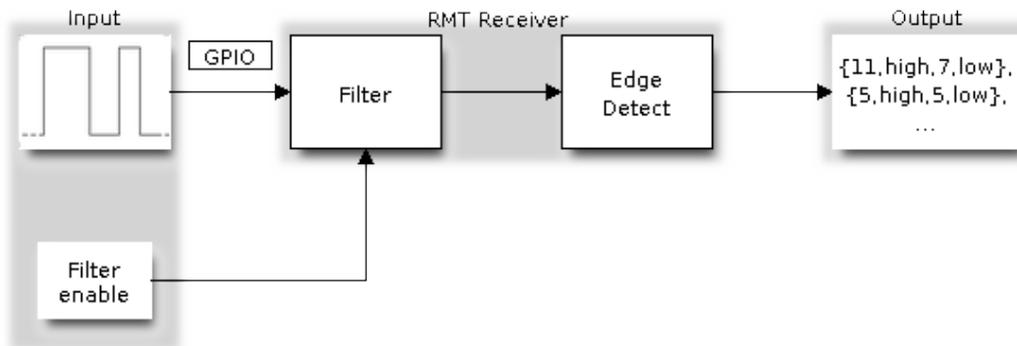


Fig. 7: RMT Receiver Overview

2. *Transmit Data or Receive Data*
3. *Change Operation Parameters*
4. *Use Interrupts*

The RMT has eight channels numbered from zero to seven. Each channel is able to independently transmit or receive data. They are referred to using indexes defined in structure `rmt_channel_t`.

## Configure Driver

There are several parameters that define how particular channel operates. Most of these parameters are configured by setting specific members of `rmt_config_t` structure. Some of the parameters are common to both transmit or receive mode, and some are mode specific. They are all discussed below.

## Common Parameters

- The **channel** to be configured, select one from the `rmt_channel_t` enumerator.
- The RMT **operation mode** - whether this channel is used to transmit or receive data, selected by setting a **rmt\_mode** members to one of the values from `rmt_mode_t`.
- What is the **pin number** to transmit or receive RMT signals, selected by setting **gpio\_num**.
- How many **memory blocks** will be used by the channel, set with **mem\_block\_num**.
- A **clock divider**, that will determine the range of pulse length generated by the RMT transmitter or discriminated by the receiver. Selected by setting **clk\_div** to a value within [1 .. 255] range. The RMT source clock is typically APB CLK, 80Mhz by default.

---

**Note:** The period of a square wave after the clock divider is called a ‘tick’. The length of the pulses generated by the RMT transmitter or discriminated by the receiver is configured in number of ‘ticks’.

---

There are also couple of specific parameters that should be set up depending if selected channel is configured in *Transmit Mode* or *Receive Mode*:

## Transmit Mode

When configuring channel in transmit mode, set **tx\_config** and the following members of `rmt_tx_config_t`:

- Transmit the currently configured data items in a loop - **loop\_en**
- Enable the RMT carrier signal - **carrier\_en**
- Frequency of the carrier in Hz - **carrier\_freq\_hz**
- Duty cycle of the carrier signal in percent (%) - **carrier\_duty\_percent**
- Level of the RMT output, when the carrier is applied - **carrier\_level**
- Enable the RMT output if idle - **idle\_output\_en**
- Set the signal level on the RMT output if idle - **idle\_level**

## Receive Mode

In receive mode, set `rx_config` and the following members of `rmt_rx_config_t`:

- Enable a filter on the input of the RMT receiver - `filter_en`
- A threshold of the filter, set in the number of ticks - `filter_ticks_thresh`. Pulses shorter than this setting will be filtered out. Note, that the range of entered tick values is [0..255].
- A pulse length threshold that will turn the RMT receiver idle, set in number of ticks - `idle_threshold`. The receiver will ignore pulses longer than this setting.

## Finalize Configuration

Once the `rmt_config_t` structure is populated with parameters, it should be then invoked with `rmt_config()` to make the configuration effective.

The last configuration step is installation of the driver in memory by calling `rmt_driver_install()`. If `rx_buf_size` parameter of this function is `> 0`, then a ring buffer for incoming data will be allocated. A default ISR handler will be installed, see a note in *Use Interrupts*.

Now, depending on how the channel is configured, we are ready to either *Transmit Data* or *Receive Data*. This is described in next two sections.

## Transmit Data

Before being able to transmit some RMT pulses, we need to define the pulse pattern. The minimum pattern recognized by the RMT controller, later called an ‘item’, is provided in a structure `rmt_item32_t`, see `soc/esp32/include/soc/rmt_struct.h`. Each item consists of two pairs of two values. The first value in a pair describes the signal duration in ticks and is 15 bits long, the second provides the signal level (high or low) and is contained in a single bit. A block of couple of items and the structure of an item is presented below.

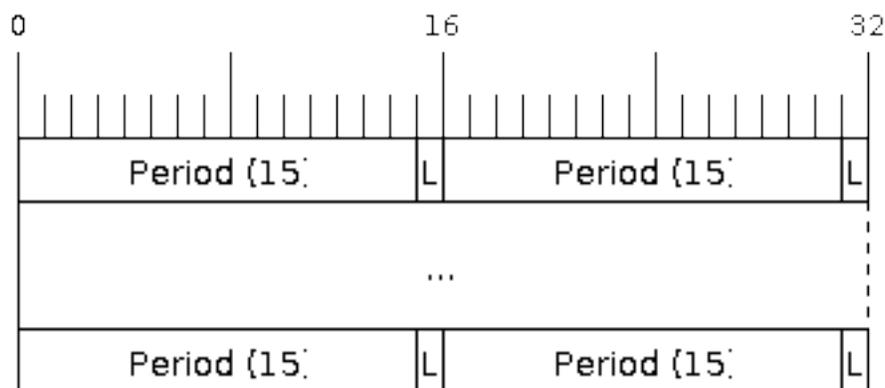


Fig. 8: Structure of RMT items (L - signal level)

For a simple example how to define a block of items see `peripherals/rmt_tx`.

The items are provided to the RMT controller by calling function `rmt_write_items()`. This function also automatically triggers start of transmission. It may be called to wait for transmission completion or exit just after transmission start. In such case you can wait for the transmission end by calling `rmt_wait_tx_done()`. This function does not limit the number of data items to transmit. It is using an interrupt to successively copy the new data chunks to RMT's internal memory as previously provided data are sent out.

Another way to provide data for transmission is by calling `rmt_fill_tx_items()`. In this case transmission is not started automatically. To control the transmission process use `rmt_tx_start()` and `rmt_tx_stop()`. The number of items to sent is restricted by the size of memory blocks allocated in the RMT controller's internal memory, see `rmt_set_mem_block_num()`.

## Receive Data

Before starting the receiver we need some storage for incoming items. The RMT controller has 512 x 32-bits of internal RAM shared between all eight channels. In typical scenarios it is not enough as an ultimate storage for all incoming (and outgoing) items. Therefore this API supports retrieval of incoming items on the fly to save them in a ring buffer of a size defined by the user. The size is provided when calling `rmt_driver_install()` discussed above. To get a handle to this buffer call `rmt_get_ringbuf_handle()`.

With the above steps complete we can start the receiver by calling `rmt_rx_start()` and then move to checking what's inside the buffer. To do so, you can use common FreeRTOS functions that interact with the ring buffer. Please see an example how to do it in `peripherals/rmt_nec_tx_rx`.

To stop the receiver, call `rmt_rx_stop()`.

## Change Operation Parameters

Previously described function `rmt_config()` provides a convenient way to set several configuration parameters in one shot. This is usually done on application start. Then, when the application is running, the API provides an alternate way to update individual parameters by calling dedicated functions. Each function refers to the specific RMT channel provided as the first input parameter. Most of the functions have `_get_` counterpart to read back the currently configured value.

## Parameters Common to Transmit and Receive Mode

- Selection of a GPIO pin number on the input or output of the RMT - `rmt_set_pin()`
- Number of memory blocks allocated for the incoming or outgoing data - `rmt_set_mem_pd()`
- Setting of the clock divider - `rmt_set_clk_div()`
- Selection of the clock source, note that currently one clock source is supported, the APB clock which is 80Mhz - `rmt_set_source_clk()`

## Transmit Mode Parameters

- Enable or disable the loop back mode for the transmitter - `rmt_set_tx_loop_mode()`
- Binary level on the output to apply the carrier - `rmt_set_tx_carrier()`, selected from `rmt_carrier_level_t`
- Determines the binary level on the output when transmitter is idle - `rmt_set_idle_level()`, selected from `rmt_idle_level_t`

## Receive Mode Parameters

- The filter setting - `rmt_set_rx_filter()`
- The receiver threshold setting - `rmt_set_rx_idle_thresh()`
- Whether the transmitter or receiver is entitled to access RMT's memory - `rmt_set_memory_owner()`, selection is from `rmt_mem_owner_t`.

## Use Interrupts

Registering of an interrupt handler for the RMT controller is done by calling `rmt_isr_register()`.

---

**Note:** When calling `rmt_driver_install()` to use the system RMT driver, a default ISR is being installed. In such a case you cannot register a generic ISR handler with `rmt_isr_register()`.

---

The RMT controller triggers interrupts on four specific events described below. To enable interrupts on these events, the following functions are provided:

- The RMT receiver has finished receiving a signal - `rmt_set_rx_intr_en()`
- The RMT transmitter has finished transmitting the signal - `rmt_set_tx_intr_en()`
- The number of events the transmitter has sent matches a threshold value `rmt_set_tx_thr_intr_en()`
- Ownership to the RMT memory block has been violated - `rmt_set_err_intr_en()`

Setting or clearing an interrupt enable mask for specific channels and events may be also done by calling `rmt_set_intr_enable_mask()` or `rmt_clr_intr_enable_mask()`.

When servicing an interrupt within an ISR, the interrupt need to explicitly cleared. To do so, set specific bits described as `RMT.int_clr.val.chN_event_name` and defined as a volatile struct in `soc/esp32/include/soc/rmt_struct.h`, where N is the RMT channel number [0, 7] and the `event_name` is one of four events described above.

If you do not need an ISR anymore, you can deregister it by calling a function `rmt_isr_deregister()`.

## Uninstall Driver

If the RMT driver has been installed with `rmt_driver_install()` for some specific period of time and then not required, the driver may be removed to free allocated resources by calling `rmt_driver_uninstall()`.

## Application Examples

- A simple RMT TX example: `peripherals/rmt_tx`.
- NEC remote control TX and RX example: `peripherals/rmt_nec_tx_rx`.

## API Reference

### Header File

- `driver/include/driver/rmt.h`

## Functions

esp\_err\_t **rmt\_set\_clk\_div**(*rmt\_channel\_t* channel, uint8\_t *div\_cnt*)

Set RMT clock divider, channel clock is divided from source clock.

### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

### Parameters

- channel: RMT channel (0-7)
- div\_cnt: RMT counter clock divider

esp\_err\_t **rmt\_get\_clk\_div**(*rmt\_channel\_t* channel, uint8\_t \**div\_cnt*)

Get RMT clock divider, channel clock is divided from source clock.

### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

### Parameters

- channel: RMT channel (0-7)
- div\_cnt: pointer to accept RMT counter divider

esp\_err\_t **rmt\_set\_rx\_idle\_thresh**(*rmt\_channel\_t* channel, uint16\_t *thresh*)

Set RMT RX idle threshold value.

In receive mode, when no edge is detected on the input signal for longer than *idle\_thres* channel clock cycles, the receive process is finished.

### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

### Parameters

- channel: RMT channel (0-7)
- thresh: RMT RX idle threshold

esp\_err\_t **rmt\_get\_rx\_idle\_thresh**(*rmt\_channel\_t* channel, uint16\_t \**thresh*)

Get RMT idle threshold value.

In receive mode, when no edge is detected on the input signal for longer than *idle\_thres* channel clock cycles, the receive process is finished.

### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

### Parameters

- `channel`: RMT channel (0-7)
- `thresh`: pointer to accept RMT RX idle threshold value

`esp_err_t rmt_set_mem_block_num(rmt_channel_t channel, uint8_t rmt_mem_num)`  
Set RMT memory block number for RMT channel.

This function is used to configure the amount of memory blocks allocated to channel `n`. The 8 channels share a 512x32-bit RAM block which can be read and written by the processor cores over the APB bus, as well as read by the transmitters and written by the receivers.

The RAM address range for channel `n` is `start_addr_CHn` to `end_addr_CHn`, which are defined by: Memory block start address is `RMT_CHANNEL_MEM(n)` (in `soc/rmt_reg.h`), that is, `start_addr_chn = RMT base address + 0x800 + 64 * n`, and `end_addr_chn = RMT base address + 0x800 + 64 * n + 64 * RMT_MEM_SIZE_CHn mod 512 * 4`

**Note** If memory block number of one channel is set to a value greater than 1, this channel will occupy the memory block of the next channel. Channel 0 can use at most 8 blocks of memory, accordingly channel 7 can only use one memory block.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0-7)
- `rmt_mem_num`: RMT RX memory block number, one block has  $64 * 32$  bits.

`esp_err_t rmt_get_mem_block_num(rmt_channel_t channel, uint8_t *rmt_mem_num)`  
Get RMT memory block number.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0-7)
- `rmt_mem_num`: Pointer to accept RMT RX memory block number

`esp_err_t rmt_set_tx_carrier(rmt_channel_t channel, bool carrier_en, uint16_t high_level, uint16_t low_level, rmt_carrier_level_t carrier_level)`  
Configure RMT carrier for TX signal.

Set different values for `carrier_high` and `carrier_low` to set different frequency of carrier. The unit of `carrier_high/low` is the source clock tick, not the divided channel counter clock.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0-7)

- `carrier_en`: Whether to enable output carrier.
- `high_level`: High level duration of carrier
- `low_level`: Low level duration of carrier.
- `carrier_level`: Configure the way carrier wave is modulated for channel 0-7.
  - 1'b1:transmit on low output level
  - 1'b0:transmit on high output level

`esp_err_t rmt_set_mem_pd(rmt_channel_t channel, bool pd_en)`  
Set RMT memory in low power mode.

Reduce power consumed by memory. 1:memory is in low power state.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0-7)
- `pd_en`: RMT memory low power enable.

`esp_err_t rmt_get_mem_pd(rmt_channel_t channel, bool *pd_en)`  
Get RMT memory low power mode.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0-7)
- `pd_en`: Pointer to accept RMT memory low power mode.

`esp_err_t rmt_tx_start(rmt_channel_t channel, bool tx_idx_rst)`  
Set RMT start sending data from memory.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0-7)
- `tx_idx_rst`: Set true to reset memory index for TX. Otherwise, transmitter will continue sending from the last index in memory.

`esp_err_t rmt_tx_stop(rmt_channel_t channel)`  
Set RMT stop sending.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- `channel`: RMT channel (0-7)

`esp_err_t rmt_rx_start` (*rmt\_channel\_t channel*, bool *rx\_idx\_rst*)  
Set RMT start receiving data.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- `channel`: RMT channel (0-7)
- `rx_idx_rst`: Set true to reset memory index for receiver. Otherwise, receiver will continue receiving data to the last index in memory.

`esp_err_t rmt_rx_stop` (*rmt\_channel\_t channel*)  
Set RMT stop receiving data.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- `channel`: RMT channel (0-7)

`esp_err_t rmt_memory_rw_rst` (*rmt\_channel\_t channel*)  
Reset RMT TX/RX memory index.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- `channel`: RMT channel (0-7)

`esp_err_t rmt_set_memory_owner` (*rmt\_channel\_t channel*, *rmt\_mem\_owner\_t owner*)  
Set RMT memory owner.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- `channel`: RMT channel (0-7)
- `owner`: To set when the transmitter or receiver can process the memory of channel.

`esp_err_t rmt_get_memory_owner (rmt_channel_t channel, rmt_mem_owner_t *owner)`  
Get RMT memory owner.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- channel: RMT channel (0-7)
- owner: Pointer to get memory owner.

`esp_err_t rmt_set_tx_loop_mode (rmt_channel_t channel, bool loop_en)`  
Set RMT tx loop mode.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- channel: RMT channel (0-7)
- loop\_en: Enable RMT transmitter loop sending mode. If set true, transmitter will continue sending from the first data to the last data in channel 0-7 over and over again in a loop.

`esp_err_t rmt_get_tx_loop_mode (rmt_channel_t channel, bool *loop_en)`  
Get RMT tx loop mode.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- channel: RMT channel (0-7)
- loop\_en: Pointer to accept RMT transmitter loop sending mode.

`esp_err_t rmt_set_rx_filter (rmt_channel_t channel, bool rx_filter_en, uint8_t thresh)`  
Set RMT RX filter.

In receive mode, channel 0-7 will ignore input pulse when the pulse width is smaller than threshold. Counted in source clock, not divided counter clock.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- channel: RMT channel (0-7)
- rx\_filter\_en: To enable RMT receiver filter.
- thresh: Threshold of pulse width for receiver.

esp\_err\_t **rmt\_set\_source\_clk** (*rmt\_channel\_t* channel, *rmt\_source\_clk\_t* base\_clk)  
Set RMT source clock.

RMT module has two clock sources:

1. APB clock which is 80Mhz
2. REF tick clock, which would be 1Mhz (not supported in this version).

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- channel: RMT channel (0-7)
- base\_clk: To choose source clock for RMT module.

esp\_err\_t **rmt\_get\_source\_clk** (*rmt\_channel\_t* channel, *rmt\_source\_clk\_t* \*src\_clk)  
Get RMT source clock.

RMT module has two clock sources:

1. APB clock which is 80Mhz
2. REF tick clock, which would be 1Mhz (not supported in this version).

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- channel: RMT channel (0-7)
- src\_clk: Pointer to accept source clock for RMT module.

esp\_err\_t **rmt\_set\_idle\_level** (*rmt\_channel\_t* channel, bool idle\_out\_en, *rmt\_idle\_level\_t* level)  
Set RMT idle output level for transmitter.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- channel: RMT channel (0-7)
- idle\_out\_en: To enable idle level output.
- level: To set the output signal's level for channel 0-7 in idle state.

esp\_err\_t **rmt\_get\_status** (*rmt\_channel\_t* channel, uint32\_t \*status)  
Get RMT status.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error

- ESP\_OK Success

**Parameters**

- `channel`: RMT channel (0-7)
- `status`: Pointer to accept channel status.

void `rmt_set_intr_enable_mask` (uint32\_t *mask*)  
Set mask value to RMT interrupt enable register.

**Parameters**

- `mask`: Bit mask to set to the register

void `rmt_clr_intr_enable_mask` (uint32\_t *mask*)  
Clear mask value to RMT interrupt enable register.

**Parameters**

- `mask`: Bit mask to clear the register

esp\_err\_t `rmt_set_rx_intr_en` (*rmt\_channel\_t* *channel*, bool *en*)  
Set RMT RX interrupt enable.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- `channel`: RMT channel (0 - 7)
- `en`: enable or disable RX interrupt.

esp\_err\_t `rmt_set_err_intr_en` (*rmt\_channel\_t* *channel*, bool *en*)  
Set RMT RX error interrupt enable.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- `channel`: RMT channel (0 - 7)
- `en`: enable or disable RX err interrupt.

esp\_err\_t `rmt_set_tx_intr_en` (*rmt\_channel\_t* *channel*, bool *en*)  
Set RMT TX interrupt enable.

**Return**

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

**Parameters**

- `channel`: RMT channel (0 - 7)
- `en`: enable or disable TX interrupt.

`esp_err_t rmt_set_tx_thr_intr_en` (*rmt\_channel\_t* channel, bool en, uint16\_t evt\_thresh)  
Set RMT TX threshold event interrupt enable.

An interrupt will be triggered when the number of transmitted items reaches the threshold value

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0 - 7)
- `en`: enable or disable TX event interrupt.
- `evt_thresh`: RMT event interrupt threshold value

`esp_err_t rmt_set_pin` (*rmt\_channel\_t* channel, *rmt\_mode\_t* mode, *gpio\_num\_t* gpio\_num)  
Set RMT pin.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0 - 7)
- `mode`: TX or RX mode for RMT
- `gpio_num`: GPIO number to transmit or receive the signal.

`esp_err_t rmt_config` (const *rmt\_config\_t* \*rmt\_param)  
Configure RMT parameters.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `rmt_param`: RMT parameter struct

`esp_err_t rmt_isr_register` (void (\*fn)) void \*  
, void \*arg, int intr\_alloc\_flags, *rmt\_isr\_handle\_t* \*handle Register RMT interrupt handler, the handler is an ISR.

The handler will be attached to the same CPU core that this function is running on.

**Note** If you already called `rmt_driver_install` to use system RMT driver, please do not register ISR handler again.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Function pointer error.
- ESP\_FAIL System driver installed, can not register ISR handler for RMT

#### Parameters

- *fn*: Interrupt handler function.
- *arg*: Parameter for the handler function
- *intr\_alloc\_flags*: Flags used to allocate the interrupt. One or multiple (ORred) ESP\_INTR\_FLAG\_\* values. See *esp\_intr\_alloc.h* for more info.
- *handle*: If non-zero, a handle to later clean up the ISR gets stored here.

`esp_err_t rmt_isr_deregister(rmt_isr_handle_t handle)`  
Deregister previously registered RMT interrupt handler.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Handle invalid

#### Parameters

- *handle*: Handle obtained from *rmt\_isr\_register*

`esp_err_t rmt_fill_tx_items(rmt_channel_t channel, const rmt_item32_t *item, uint16_t item_num, uint16_t mem_offset)`  
Fill memory data of channel with given RMT items.

#### Return

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- *channel*: RMT channel (0 - 7)
- *item*: Pointer of items.
- *item\_num*: RMT sending items number.
- *mem\_offset*: Index offset of memory.

`esp_err_t rmt_driver_install(rmt_channel_t channel, size_t rx_buf_size, int intr_alloc_flags)`  
Initialize RMT driver.

#### Return

- ESP\_ERR\_INVALID\_STATE Driver is already installed, call *rmt\_driver\_uninstall* first.
- ESP\_ERR\_NO\_MEM Memory allocation failure
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_OK Success

#### Parameters

- *channel*: RMT channel (0 - 7)

- `rx_buf_size`: Size of RMT RX ringbuffer. Can be 0 if the RX ringbuffer is not used.
- `intr_alloc_flags`: Flags for the RMT driver interrupt handler. Pass 0 for default flags. See `esp_intr_alloc.h` for details.

`esp_err_t rmt_driver_uninstall (rmt_channel_t channel)`  
 Uninstall RMT driver.

**Return**

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

**Parameters**

- `channel`: RMT channel (0 - 7)

`esp_err_t rmt_write_items (rmt_channel_t channel, const rmt_item32_t *rmt_item, int item_num, bool wait_tx_done)`  
 RMT send waveform from `rmt_item` array.

This API allows user to send waveform with any length.

**Note** This function will not copy data, instead, it will point to the original items, and send the waveform items. If `wait_tx_done` is set to true, this function will block and will not return until all items have been sent out. If `wait_tx_done` is set to false, this function will return immediately, and the driver interrupt will continue sending the items. We must make sure the item data will not be damaged when the driver is still sending items in driver interrupt.

**Return**

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

**Parameters**

- `channel`: RMT channel (0 - 7)
- `rmt_item`: head point of RMT items array.
- `item_num`: RMT data item number.
- `wait_tx_done`:
  - If set 1, it will block the task and wait for sending done.
  - If set 0, it will not wait and return immediately.

`esp_err_t rmt_wait_tx_done (rmt_channel_t channel, TickType_t wait_time)`  
 Wait RMT TX finished.

**Return**

- `ESP_OK` RMT Tx done successfully
- `ESP_ERR_TIMEOUT` Exceeded the ‘`wait_time`’ given
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_FAIL` Driver not installed

**Parameters**

- `channel`: RMT channel (0 - 7)
- `wait_time`: Maximum time in ticks to wait for transmission to be complete

`esp_err_t rmt_get_ringbuf_handle` (*rmt\_channel\_t* channel, *RingbufHandle\_t* \*buf\_handle)  
Get ringbuffer from RMT.

Users can get the RMT RX ringbuffer handle, and process the RX data.

#### Return

- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_OK` Success

#### Parameters

- `channel`: RMT channel (0 - 7)
- `buf_handle`: Pointer to buffer handle to accept RX ringbuffer handle.

## Structures

**struct** `rmt_tx_config_t`

Data struct of RMT TX configure parameters.

#### Public Members

bool `loop_en`

Enable sending RMT items in a loop

uint32\_t `carrier_freq_hz`

RMT carrier frequency

uint8\_t `carrier_duty_percent`

RMT carrier duty (%)

*rmt\_carrier\_level\_t* `carrier_level`

Level of the RMT output, when the carrier is applied

bool `carrier_en`

RMT carrier enable

*rmt\_idle\_level\_t* `idle_level`

RMT idle level

bool `idle_output_en`

RMT idle level output enable

**struct** `rmt_rx_config_t`

Data struct of RMT RX configure parameters.

#### Public Members

bool `filter_en`

RMT receiver filter enable

uint8\_t `filter_ticks_thresh`

RMT filter tick number

`uint16_t idle_threshold`  
RMT RX idle threshold

**struct rmt\_config\_t**  
Data struct of RMT configure parameters.

### Public Members

*rmt\_mode\_t* **rmt\_mode**  
RMT mode: transmitter or receiver

*rmt\_channel\_t* **channel**  
RMT channel

`uint8_t clk_div`  
RMT channel counter divider

*gpio\_num\_t* **gpio\_num**  
RMT GPIO number

`uint8_t mem_block_num`  
RMT memory block number

*rmt\_tx\_config\_t* **tx\_config**  
RMT TX parameter

*rmt\_rx\_config\_t* **rx\_config**  
RMT RX parameter

### Macros

**RMT\_MEM\_BLOCK\_BYTE\_NUM**

**RMT\_MEM\_ITEM\_NUM**

### Type Definitions

**typedef intr\_handle\_t rmt\_isr\_handle\_t**

### Enumerations

**enum rmt\_channel\_t**

*Values:*

**RMT\_CHANNEL\_0** = 0  
RMT Channel 0

**RMT\_CHANNEL\_1**  
RMT Channel 1

**RMT\_CHANNEL\_2**  
RMT Channel 2

**RMT\_CHANNEL\_3**  
RMT Channel 3

**RMT\_CHANNEL\_4**  
RMT Channel 4

**RMT\_CHANNEL\_5**  
RMT Channel 5

**RMT\_CHANNEL\_6**  
RMT Channel 6

**RMT\_CHANNEL\_7**  
RMT Channel 7

**RMT\_CHANNEL\_MAX**

**enum rmt\_mem\_owner\_t**

*Values:*

**RMT\_MEM\_OWNER\_TX** = 0  
RMT RX mode, RMT transmitter owns the memory block

**RMT\_MEM\_OWNER\_RX** = 1  
RMT RX mode, RMT receiver owns the memory block

**RMT\_MEM\_OWNER\_MAX**

**enum rmt\_source\_clk\_t**

*Values:*

**RMT\_BASECLK\_REF** = 0  
RMT source clock system reference tick, 1MHz by default (not supported in this version)

**RMT\_BASECLK\_APB**  
RMT source clock is APB CLK, 80Mhz by default

**RMT\_BASECLK\_MAX**

**enum rmt\_data\_mode\_t**

*Values:*

**RMT\_DATA\_MODE\_FIFO** = 0

**RMT\_DATA\_MODE\_MEM** = 1

**RMT\_DATA\_MODE\_MAX**

**enum rmt\_mode\_t**

*Values:*

**RMT\_MODE\_TX** = 0  
RMT TX mode

**RMT\_MODE\_RX**  
RMT RX mode

**RMT\_MODE\_MAX**

**enum rmt\_idle\_level\_t**

*Values:*

**RMT\_IDLE\_LEVEL\_LOW** = 0  
RMT TX idle level: low Level

**RMT\_IDLE\_LEVEL\_HIGH**  
RMT TX idle level: high Level

**RMT\_IDLE\_LEVEL\_MAX**

**enum rmt\_carrier\_level\_t**

*Values:*

**RMT\_CARRIER\_LEVEL\_LOW** = 0

RMT carrier wave is modulated for low Level output

**RMT\_CARRIER\_LEVEL\_HIGH**

RMT carrier wave is modulated for high Level output

**RMT\_CARRIER\_LEVEL\_MAX**

## 2.4.10 SDMMC Host Peripheral

### Overview

SDMMC peripheral supports SD and MMC memory cards and SDIO cards. SDMMC software builds on top of SDMMC driver and consists of the following parts:

1. SDMMC host driver (`driver/sdmmc_host.h`) — this driver provides APIs to send commands to the slave device(s), send and receive data, and handling error conditions on the bus.
2. SDMMC protocol layer (`sdmmc_cmd.h`) — this component handles specifics of SD protocol such as card initialization and data transfer commands. Despite the name, only SD (SDSC/SDHC/SDXC) cards are supported at the moment. Support for MCC/eMMC cards can be added in the future.

Protocol layer works with the host via `sdmmc_host_t` structure. This structure contains pointers to various functions of the host.

In addition to SDMMC Host peripheral, ESP32 has SPI peripherals which can also be used to work with SD cards. This is supported using a variant of the host driver, `driver/sdspi_host.h`. This driver has the same interface as SDMMC host driver, and the protocol layer can use either of two.

### Application Example

An example which combines SDMMC driver with FATFS library is provided in `examples/storage/sd_card` directory. This example initializes the card, writes and reads data from it using POSIX and C library APIs. See `README.md` file in the example directory for more information.

### Protocol layer APIs

Protocol layer is given `sdmmc_host_t` structure which describes the SD/MMC host driver, lists its capabilities, and provides pointers to functions of the driver. Protocol layer stores card-specific information in `sdmmc_card_t` structure. When sending commands to the SD/MMC host driver, protocol layer uses `sdmmc_command_t` structure to describe the command, argument, expected return value, and data to transfer, if any.

Normal usage of the protocol layer is as follows:

1. Call the host driver functions to initialize the host (e.g. `sdmmc_host_init`, `sdmmc_host_init_slot`).
2. Call `sdmmc_card_init` to initialize the card, passing it host driver information (`host`) and a pointer to `sdmmc_card_t` structure which will be filled in (`card`).
3. To read and write sectors of the card, use `sdmmc_read_sectors` and `sdmmc_write_sectors`, passing the pointer to card information structure (`card`).
4. When card is not used anymore, call the host driver function to disable SDMMC host peripheral and free resources allocated by the driver (e.g. `sdmmc_host_deinit`).

Most applications need to use the protocol layer only in one task; therefore the protocol layer doesn't implement any kind of locking on the `sdmhc_card_t` structure, or when accessing SDMMC host driver. Such locking has to be implemented in the higher layer, if necessary (e.g. in the filesystem driver).

**struct sdmhc\_host\_t**

SD/MMC Host description

This structure defines properties of SD/MMC host and functions of SD/MMC host which can be used by upper layers.

**Public Members**

**uint32\_t flags**

flags defining host properties

**int slot**

slot number, to be passed to host functions

**int max\_freq\_khz**

max frequency supported by the host

**float io\_voltage**

I/O voltage used by the controller (voltage switching is not supported)

**esp\_err\_t (\*init) (void)**

Host function to initialize the driver

**esp\_err\_t (\*set\_bus\_width) (int slot, size\_t width)**

host function to set bus width

**esp\_err\_t (\*set\_card\_clk) (int slot, uint32\_t freq\_khz)**

host function to set card clock frequency

**esp\_err\_t (\*do\_transaction) (int slot, *sdmmc\_command\_t* \*cmdinfo)**

host function to do a transaction

**esp\_err\_t (\*deinit) (void)**

host function to deinitialize the driver

**int command\_timeout\_ms**

timeout, in milliseconds, of a single command. Set to 0 to use the default value.

**SDMMC\_HOST\_FLAG\_1BIT**

host supports 1-line SD and MMC protocol

**SDMMC\_HOST\_FLAG\_4BIT**

host supports 4-line SD and MMC protocol

**SDMMC\_HOST\_FLAG\_8BIT**

host supports 8-line MMC protocol

**SDMMC\_HOST\_FLAG\_SPI**

host supports SPI protocol

**SDMMC\_FREQ\_DEFAULT**

SD/MMC Default speed (limited by clock divider)

**SDMMC\_FREQ\_HIGHSPEED**

SD High speed (limited by clock divider)

**SDMMC\_FREQ\_PROBING**

SD/MMC probing speed

**struct sdmmc\_command\_t**  
SD/MMC command information

### Public Members

`uint32_t opcode`  
SD or MMC command index

`uint32_t arg`  
SD/MMC command argument

`sdmmc_response_t response`  
response buffer

`void *data`  
buffer to send or read into

`size_t datalen`  
length of data buffer

`size_t blklen`  
block length

`int flags`  
see below

`esp_err_t error`  
error returned from transfer

`int timeout_ms`  
response timeout, in milliseconds

**struct sdmmc\_card\_t**  
SD/MMC card information structure

### Public Members

`sdmmc_host_t host`  
Host with which the card is associated

`uint32_t ocr`  
OCR (Operation Conditions Register) value

`sdmmc_cid_t cid`  
decoded CID (Card IDentification) register value

`sdmmc_csd_t csd`  
decoded CSD (Card-Specific Data) register value

`sdmmc_scr_t scr`  
decoded SCR (SD card Configuration Register) value

`uint16_t rca`  
RCA (Relative Card Address)

**struct sdmmc\_csd\_t**  
Decoded values from SD card Card Specific Data register

### Public Members

int **csd\_ver**  
CSD structure format

int **mmc\_ver**  
MMC version (for CID format)

int **capacity**  
total number of sectors

int **sector\_size**  
sector size in bytes

int **read\_block\_len**  
block length for reads

int **card\_command\_class**  
Card Command Class for SD

int **tr\_speed**  
Max transfer speed

**struct sdmmc\_cid\_t**  
Decoded values from SD card Card IDentification register

### Public Members

int **mfg\_id**  
manufacturer identification number

int **oem\_id**  
OEM/product identification number

char **name**[8]  
product name (MMC v1 has the longest)

int **revision**  
product revision

int **serial**  
product serial number

int **date**  
manufacturing date

**struct sdmmc\_scr\_t**  
Decoded values from SD Configuration Register

### Public Members

int **sd\_spec**  
SD Physical layer specification version, reported by card

int **bus\_width**  
bus widths supported by card: BIT(0) — 1-bit bus, BIT(2) — 4-bit bus

esp\_err\_t **sdmmc\_card\_init** (**const** *sdmmc\_host\_t* \*host, *sdmmc\_card\_t* \*out\_card)  
Probe and initialize SD/MMC card using given host

**Note** Only SD cards (SDSC and SDHC/SDXC) are supported now. Support for MMC/eMMC cards will be added later.

#### Return

- ESP\_OK on success
- One of the error codes from SDMMC host controller

#### Parameters

- `host`: pointer to structure defining host controller
- `out_card`: pointer to structure which will receive information about the card when the function completes

`esp_err_t sdmmc_write_sectors` (*sdmmc\_card\_t* \*`card`, `const` void \*`src`, `size_t` `start_sector`, `size_t` `sector_count`)

Write given number of sectors to SD/MMC card

#### Return

- ESP\_OK on success
- One of the error codes from SDMMC host controller

#### Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `src`: pointer to data buffer to read data from; data size must be equal to `sector_count * card->csd.sector_size`
- `start_sector`: sector where to start writing
- `sector_count`: number of sectors to write

`esp_err_t sdmmc_read_sectors` (*sdmmc\_card\_t* \*`card`, void \*`dst`, `size_t` `start_sector`, `size_t` `sector_count`)

Write given number of sectors to SD/MMC card

#### Return

- ESP\_OK on success
- One of the error codes from SDMMC host controller

#### Parameters

- `card`: pointer to card information structure previously initialized using `sdmmc_card_init`
- `dst`: pointer to data buffer to write into; buffer size must be at least `sector_count * card->csd.sector_size`
- `start_sector`: sector where to start reading
- `sector_count`: number of sectors to read

## SDMMC host driver APIs

On the ESP32, SDMMC host peripheral has two slots:

- Slot 0 (`SDMMC_HOST_SLOT_0`) is an 8-bit slot. It uses `HS1_*` signals in the PIN MUX.
- Slot 1 (`SDMMC_HOST_SLOT_1`) is a 4-bit slot. It uses `HS2_*` signals in the PIN MUX.

Card Detect and Write Protect signals can be routed to arbitrary pins using GPIO matrix. To use these pins, set `gpio_cd` and `gpio_wp` members of `sdmmc_slot_config_t` structure when calling `sdmmc_host_init_slot`.

Of all the functions listed below, only `sdmmc_host_init`, `sdmmc_host_init_slot`, and `sdmmc_host_deinit` will be used directly by most applications. Other functions, such as `sdmmc_host_set_bus_width`, `sdmmc_host_set_card_clk`, and `sdmmc_host_do_transaction` will be called by the SD/MMC protocol layer via function pointers in `sdmmc_host_t` structure.

`esp_err_t sdmmc_host_init ()`  
Initialize SDMMC host peripheral.

**Note** This function is not thread safe

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if `sdmmc_host_init` was already called
- ESP\_ERR\_NO\_MEM if memory can not be allocated

**SDMMC\_HOST\_SLOT\_0**  
SDMMC slot 0.

**SDMMC\_HOST\_SLOT\_1**  
SDMMC slot 1.

**SDMMC\_HOST\_DEFAULT ()**  
Default `sdmmc_host_t` structure initializer for SDMMC peripheral.

Uses SDMMC peripheral, with 4-bit mode enabled, and max frequency set to 20MHz

**SDMMC\_SLOT\_WIDTH\_DEFAULT**  
use the default width for the slot (8 for slot 0, 4 for slot 1)

`esp_err_t sdmmc_host_init_slot (int slot, const sdmmc_slot_config_t *slot_config)`  
Initialize given slot of SDMMC peripheral.

On the ESP32, SDMMC peripheral has two slots:

- Slot 0: 8-bit wide, maps to HS1\_\* signals in PIN MUX
- Slot 1: 4-bit wide, maps to HS2\_\* signals in PIN MUX

Card detect and write protect signals can be routed to arbitrary GPIOs using GPIO matrix.

**Note** This function is not thread safe

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if host has not been initialized using `sdmmc_host_init`

**Parameters**

- `slot`: slot number (SDMMC\_HOST\_SLOT\_0 or SDMMC\_HOST\_SLOT\_1)
- `slot_config`: additional configuration for the slot

**struct sdmmc\_slot\_config\_t**  
Extra configuration for SDMMC peripheral slot

## Public Members

*gpio\_num\_t* **gpio\_cd**  
GPIO number of card detect signal.

*gpio\_num\_t* **gpio\_wp**  
GPIO number of write protect signal.

*uint8\_t* **width**  
Bus width used by the slot (might be less than the max width supported)

**SDMMC\_SLOT\_NO\_CD**  
indicates that card detect line is not used

**SDMMC\_SLOT\_NO\_WP**  
indicates that write protect line is not used

**SDMMC\_SLOT\_CONFIG\_DEFAULT** ()  
Macro defining default configuration of SDMMC host slot

*esp\_err\_t* **sdmmc\_host\_set\_bus\_width** (*int slot*, *size\_t width*)  
Select bus width to be used for data transfer.

SD/MMC card must be initialized prior to this command, and a command to set bus width has to be sent to the card (e.g. SD\_APP\_SET\_BUS\_WIDTH)

**Note** This function is not thread safe

### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if slot number or width is not valid

### Parameters

- *slot*: slot number (SDMMC\_HOST\_SLOT\_0 or SDMMC\_HOST\_SLOT\_1)
- *width*: bus width (1, 4, or 8 for slot 0; 1 or 4 for slot 1)

*esp\_err\_t* **sdmmc\_host\_set\_card\_clk** (*int slot*, *uint32\_t freq\_khz*)  
Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

**Note** This function is not thread safe

### Return

- ESP\_OK on success
- other error codes may be returned in the future

### Parameters

- *slot*: slot number (SDMMC\_HOST\_SLOT\_0 or SDMMC\_HOST\_SLOT\_1)
- *freq\_khz*: card clock frequency, in kHz

*esp\_err\_t* **sdmmc\_host\_do\_transaction** (*int slot*, *sdmmc\_command\_t \*cmdinfo*)  
Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

**Note** This function is not thread safe w.r.t. init/deinit functions, and bus width/clock speed configuration functions. Multiple tasks can call `sdmmc_host_do_transaction` as long as other `sdmmc_host_*` functions are not called.

**Attention** Data buffer passed in `cmdinfo->data` must be in DMA capable memory

**Return**

- `ESP_OK` on success
- `ESP_ERR_TIMEOUT` if response or data transfer has timed out
- `ESP_ERR_INVALID_CRC` if response or data transfer CRC check has failed
- `ESP_ERR_INVALID_RESPONSE` if the card has sent an invalid response
- `ESP_ERR_INVALID_SIZE` if the size of data transfer is not valid in SD protocol
- `ESP_ERR_INVALID_ARG` if the data buffer is not in DMA capable memory

**Parameters**

- `slot`: slot number (`SDMMC_HOST_SLOT_0` or `SDMMC_HOST_SLOT_1`)
- `cmdinfo`: pointer to structure describing command and data to transfer

`esp_err_t sdmmc_host_deinit ()`  
Disable SDMMC host and release allocated resources.

**Note** This function is not thread safe

**Return**

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if `sdmmc_host_init` function has not been called

## SD SPI driver APIs

SPI controllers accessible via `spi_master` driver (HSPI, VSPI) can be used to work with SD cards. In SPI mode, SD driver has lower throughput than in 1-line SD mode. However SPI mode makes pin selection more flexible, as SPI peripheral can be connected to any ESP32 pins using GPIO Matrix. SD SPI driver uses software controlled CS signal. Currently SD SPI driver assumes that it can use the SPI controller exclusively, so applications which need to share SPI bus between SD cards and other peripherals need to make sure that SD card and other devices are not used at the same time from different tasks.

SD SPI driver is represented using an `sdmmc_host_t` structure initialized using `SDSPI_HOST_DEFAULT` macro. For slot initialization, `SDSPI_SLOT_CONFIG_DEFAULT` can be used to fill in default pin mapping, which is the same as the pin mapping in SD mode.

SD SPI driver APIs are very similar to SDMMC host APIs. As with the SDMMC host driver, only `sdspi_host_init`, `sdspi_host_init_slot`, and `sdspi_host_deinit` functions are normally used by the applications. Other functions are called by the protocol level driver via function pointers in `sdmmc_host_t` structure.

`esp_err_t sdspi_host_init ()`  
Initialize SD SPI driver.

**Note** This function is not thread safe

**Return**

- `ESP_OK` on success

- other error codes may be returned in future versions

#### **SDSPI\_HOST\_DEFAULT** ( )

Default *sdmmc\_host\_t* structure initializer for SD over SPI driver.

Uses SPI mode and max frequency set to 20MHz

'slot' can be set to one of HSPI\_HOST, VSPI\_HOST.

`esp_err_t sds_pi_host_init_slot` (int *slot*, **const** *sds\_pi\_slot\_config\_t* \**slot\_config*)

Initialize SD SPI driver for the specific SPI controller.

**Note** This function is not thread safe

#### **Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if `sds_pi_init_slot` has invalid arguments
- ESP\_ERR\_NO\_MEM if memory can not be allocated
- other errors from the underlying `spi_master` and `gpio` drivers

#### **Parameters**

- `slot`: SPI controller to use (HSPI\_HOST or VSPI\_HOST)
- `slot_config`: pointer to slot configuration structure

**struct sds\_pi\_slot\_config\_t**

Extra configuration for SPI host

#### **Public Members**

*gpio\_num\_t* **gpio\_miso**  
GPIO number of MISO signal.

*gpio\_num\_t* **gpio\_mosi**  
GPIO number of MOSI signal.

*gpio\_num\_t* **gpio\_sck**  
GPIO number of SCK signal.

*gpio\_num\_t* **gpio\_cs**  
GPIO number of CS signal.

*gpio\_num\_t* **gpio\_cd**  
GPIO number of card detect signal.

*gpio\_num\_t* **gpio\_wp**  
GPIO number of write protect signal.

int **dma\_channel**  
DMA channel to be used by SPI driver (1 or 2)

**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\_CONFIG\_DEFAULT()**

Macro defining default configuration of SPI host

`esp_err_t sdspi_host_set_card_clk` (int *slot*, uint32\_t *freq\_khz*)

Set card clock frequency.

Currently only integer fractions of 40MHz clock can be used. For High Speed cards, 40MHz can be used. For Default Speed cards, 20MHz can be used.

**Note** This function is not thread safe

**Return**

- ESP\_OK on success
- other error codes may be returned in the future

**Parameters**

- *slot*: SPI controller (HSPI\_HOST or VSPI\_HOST)
- *freq\_khz*: card clock frequency, in kHz

`esp_err_t sdspi_host_do_transaction` (int *slot*, *sdmmc\_command\_t* \**cmdinfo*)

Send command to the card and get response.

This function returns when command is sent and response is received, or data is transferred, or timeout occurs.

**Note** This function is not thread safe w.r.t. `init/deinit` functions, and bus width/clock speed configuration functions. Multiple tasks can call `sdspi_host_do_transaction` as long as other `sdspi_host_*` functions are not called.

**Return**

- ESP\_OK on success
- ESP\_ERR\_TIMEOUT if response or data transfer has timed out
- ESP\_ERR\_INVALID\_CRC if response or data transfer CRC check has failed
- ESP\_ERR\_INVALID\_RESPONSE if the card has sent an invalid response

**Parameters**

- *slot*: SPI controller (HSPI\_HOST or VSPI\_HOST)
- *cmdinfo*: pointer to structure describing command and data to transfer

`esp_err_t sdspi_host_deinit` ()

Release resources allocated using `sdspi_host_init`.

**Note** This function is not thread safe

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if `sdspi_host_init` function has not been called

## 2.4.11 Sigma-delta Modulation

### Introduction

ESP32 has a second-order sigma-delta modulation module. This driver configures the channels of the sigma-delta module.

### Functionality Overview

There are eight independent sigma-delta modulation channels identified with *sigmadelta\_channel\_t*. Each channel is capable to output the binary, hardware generated signal with the sigma-delta modulation.

Selected channel should be set up by providing configuration parameters in *sigmadelta\_config\_t* and then applying this configuration with *sigmadelta\_config()*.

Another option is to call individual functions, that will configure all required parameters one by one:

- **Prescaler** of the sigma-delta generator - *sigmadelta\_set\_prescale()*
- **Duty** of the output signal - *sigmadelta\_set\_duty()*
- **GPIO pin** to output modulated signal - *sigmadelta\_set\_pin()*

The range of the 'duty' input parameter of *sigmadelta\_set\_duty()* is from -128 to 127 (eight bit signed integer). If zero value is set, then the output signal's duty will be about 50%, see description of *sigmadelta\_set\_duty()*.

### Application Example

Sigma-delta Modulation example: [peripherals/sigmadelta](#).

### API Reference

#### Header File

- `driver/include/driver/sigmadelta.h`

#### Functions

`esp_err_t sigmadelta_config(const sigmadelta_config_t *config)`

Configure Sigma-delta channel.

#### Return

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error

#### Parameters

- `config`: Pointer of Sigma-delta channel configuration struct

`esp_err_t sigmadelta_set_duty (sigmadelta_channel_t channel, int8_t duty)`  
Set Sigma-delta channel duty.

This function is used to set Sigma-delta channel duty, If you add a capacitor between the output pin and ground, the average output voltage will be  $V_{dc} = VDDIO / 256 * duty + VDDIO/2$ , where VDDIO is the power supply voltage.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `channel`: Sigma-delta channel number
- `duty`: Sigma-delta duty of one channel, the value ranges from -128 to 127, recommended range is -90 ~ 90. The waveform is more like a random one in this range.

`esp_err_t sigmadelta_set_prescale (sigmadelta_channel_t channel, uint8_t prescale)`

Set Sigma-delta channel's clock pre-scale value. The source clock is APP\_CLK, 80MHz. The clock frequency of the sigma-delta channel is APP\_CLK / pre\_scale.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `channel`: Sigma-delta channel number
- `prescale`: The divider of source clock, ranges from 0 to 255

`esp_err_t sigmadelta_set_pin (sigmadelta_channel_t channel, gpio_num_t gpio_num)`

Set Sigma-delta signal output pin.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `channel`: Sigma-delta channel number
- `gpio_num`: GPIO number of output pin.

## Structures

`struct sigmadelta_config_t`

Sigma-delta configure struct.

## Public Members

*sigmadelta\_channel\_t* **channel**

Sigma-delta channel number

*int8\_t* **sigmadelta\_duty**

Sigma-delta duty, duty ranges from -128 to 127.

*uint8\_t* **sigmadelta\_prescale**

Sigma-delta prescale, prescale ranges from 0 to 255.

*uint8\_t* **sigmadelta\_gpio**

Sigma-delta output io number, refer to gpio.h for more details.

## Enumerations

**enum sigmadelta\_channel\_t**

Sigma-delta channel list.

*Values:*

**SIGMADELTA\_CHANNEL\_0** = 0

Sigma-delta channel 0

**SIGMADELTA\_CHANNEL\_1** = 1

Sigma-delta channel 1

**SIGMADELTA\_CHANNEL\_2** = 2

Sigma-delta channel 2

**SIGMADELTA\_CHANNEL\_3** = 3

Sigma-delta channel 3

**SIGMADELTA\_CHANNEL\_4** = 4

Sigma-delta channel 4

**SIGMADELTA\_CHANNEL\_5** = 5

Sigma-delta channel 5

**SIGMADELTA\_CHANNEL\_6** = 6

Sigma-delta channel 6

**SIGMADELTA\_CHANNEL\_7** = 7

Sigma-delta channel 7

**SIGMADELTA\_CHANNEL\_MAX**

## 2.4.12 SPI Master driver

### Overview

The ESP32 has four SPI peripheral devices, called SPI0, SPI1, HSPI and VSPI. SPI0 is entirely dedicated to the flash cache the ESP32 uses to map the SPI flash device it is connected to into memory. SPI1 is connected to the same hardware lines as SPI0 and is used to write to the flash chip. HSPI and VSPI are free to use. SPI1, HSPI and VSPI all have three chip select lines, allowing them to drive up to three SPI devices each as a master.

## The spi\_master driver

The spi\_master driver allows easy communicating with SPI slave devices, even in a multithreaded environment. It fully transparently handles DMA transfers to read and write data and automatically takes care of multiplexing between different SPI slaves on the same master

## Terminology

The spi\_master driver uses the following terms:

- **Host:** The SPI peripheral inside the ESP32 initiating the SPI transmissions. One of SPI, HSPI or VSPI. (For now, only HSPI or VSPI are actually supported in the driver; it will support all 3 peripherals somewhere in the future.)
- **Bus:** The SPI bus, common to all SPI devices connected to one host. In general the bus consists of the miso, mosi, sclk and optionally quadwp and quadhd signals. The SPI slaves are connected to these signals in parallel.
  - miso - Also known as q, this is the input of the serial stream into the ESP32
  - mosi - Also known as d, this is the output of the serial stream from the ESP32
  - sclk - Clock signal. Each data bit is clocked out or in on the positive or negative edge of this signal
  - quadwp - Write Protect signal. Only used for 4-bit (qio/qout) transactions.
  - quadhd - Hold signal. Only used for 4-bit (qio/qout) transactions.
- **Device:** A SPI slave. Each SPI slave has its own chip select (CS) line, which is made active when a transmission to/from the SPI slave occurs.
- **Transaction:** One instance of CS going active, data transfer from and/or to a device happening, and CS going inactive again. Transactions are atomic, as in they will never be interrupted by another transaction.

## SPI transactions

A transaction on the SPI bus consists of five phases, any of which may be skipped:

- The command phase. In this phase, a command (0-16 bit) is clocked out.
- The address phase. In this phase, an address (0-64 bit) is clocked out.
- The write phase. The master sends data to the slave.
- The dummy phase. The phase is configurable, used to meet the timing requirements.
- The read phase. The slave sends data to the master.

In full duplex, the read and write phases are combined, causing the SPI host to read and write data simultaneously. The total transaction length is decided by `command_bits + address_bits + trans_conf.length`, while the `trans_conf.rx_length` only determines length of data received into the buffer.

In half duplex, the length of write phase and read phase are decided by `trans_conf.length` and `trans_conf.rx_length` respectively. **\*\* Note that a half duplex transaction with both a read and write phase is not supported when using DMA. \*\*** If such transaction is needed, you have to use one of the alternative solutions:

1. use full-duplex mode instead.
2. disable the DMA by set the last parameter to 0 in bus initialization function just as belows:  

```
ret=spi_bus_initialize(VSPI_HOST, &buscfg, 0);
```

  
this may prohibit you from transmitting and receiving data longer than 32 bytes.

3. try to use command and address field to replace the write phase.

The command and address phase are optional in that not every SPI device will need to be sent a command and/or address. This is reflected in the device configuration: when the `command_bits` or `address_bits` fields are set to zero, no command or address phase is done.

Something similar is true for the read and write phase: not every transaction needs both data to be written as well as data to be read. When `rx_buffer` is NULL (and `SPI_USE_RXDATA`) is not set) the read phase is skipped. When `tx_buffer` is NULL (and `SPI_USE_TXDATA`) is not set) the write phase is skipped.

### Using the `spi_master` driver

- Initialize a SPI bus by calling `spi_bus_initialize`. Make sure to set the correct IO pins in the `bus_config` struct. Take care to set signals that are not needed to -1.
- Tell the driver about a SPI slave device connected to the bus by calling `spi_bus_add_device`. Make sure to configure any timing requirements the device has in the `dev_config` structure. You should now have a handle for the device, to be used when sending it a transaction.
- To interact with the device, fill one or more `spi_transaction_t` structure with any transaction parameters you need. Either queue all transactions by calling `spi_device_queue_trans`, later quering the result using `spi_device_get_trans_result`, or handle all requests synchronously by feeding them into `spi_device_transmit`.
- Optional: to unload the driver for a device, call `spi_bus_remove_device` with the device handle as an argument
- Optional: to remove the driver for a bus, make sure no more drivers are attached and call `spi_bus_free`.

### Command and address phases

During the command and address phases, `cmd` and `addr` field in the `spi_transaction_t` struct are sent to the bus, while nothing is read at the same time. The default length of command and address phase are set in the `spi_device_interface_config_t` and by `spi_bus_add_device`. When the the flag `SPI_TRANS_VARIABLE_CMD` and `SPI_TRANS_VARIABLE_ADDR` are not set in the `spi_transaction_t`, the driver automatically set the length of these phases to the default value as set when the device is initialized respectively.

If the length of command and address phases needs to be variable, declare a `spi_transaction_ext_t` descriptor, set the flag `SPI_TRANS_VARIABLE_CMD` or/and `SPI_TRANS_VARIABLE_ADDR` in the `flags` of base member and configure the rest part of base as usual. Then the length of each phases will be `command_bits` and `address_bits` set in the `spi_transaction_ext_t`.

### Write and read phases

Normally, data to be transferred to or from a device will be read from or written to a chunk of memory indicated by the `rx_buffer` and `tx_buffer` members of the transaction structure. When DMA is enabled for transfers, these buffers are highly recommended to meet the requirements as follows:

1. allocated in DMA-capable memory using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`;
2. 32-bit aligned (start from the boundary and have length of multiples of 4 bytes).

If these requirements are not satisfied, efficiency of the transaction will suffer due to the allocation and memcopy of temporary buffers.

Sometimes, the amount of data is very small making it less than optimal allocating a separate buffer for it. If the data to be transferred is 32 bits or less, it can be stored in the transaction struct itself. For transmitted data, use the `tx_data` member for this and set the `SPI_USE_TXDATA` flag on the transmission. For received data, use `rx_data` and set `SPI_USE_RXDATA`. In both cases, do not touch the `tx_buffer` or `rx_buffer` members, because they use the same memory locations as `tx_data` and `rx_data`.

## Application Example

Display graphics on the 320x240 LCD of WROVER-Kits: [peripherals/spi\\_master](#).

## API Reference - SPI Common

### Header File

- `driver/include/driver/spi_common.h`

### Functions

bool **spicommon\_periph\_claim** (*spi\_host\_device\_t* host)

Try to claim a SPI peripheral.

Call this if your driver wants to manage a SPI peripheral.

**Return** True if peripheral is claimed successfully; false if peripheral already is claimed.

#### Parameters

- `host`: Peripheral to claim

bool **spicommon\_periph\_free** (*spi\_host\_device\_t* host)

Return the SPI peripheral so another driver can claim it.

**Return** True if peripheral is returned successfully; false if peripheral was free to claim already.

#### Parameters

- `host`: Peripheral to return

bool **spicommon\_dma\_chan\_claim** (int *dma\_chan*)

Try to claim a SPI DMA channel.

Call this if your driver wants to use SPI with a DMA channel.

**Return** True if success; false otherwise.

#### Parameters

- `dma_chan`: channel to claim

bool **spicommon\_dma\_chan\_free** (int *dma\_chan*)

Return the SPI DMA channel so other driver can claim it, or just to power down DMA.

**Return** True if success; false otherwise.

#### Parameters

- `dma_chan`: channel to return

`esp_err_t spicommon_bus_initialize_io` (*spi\_host\_device\_t* host, **const** *spi\_bus\_config\_t* \*bus\_config, int dma\_chan, int flags, bool \*is\_native)

Connect a SPI peripheral to GPIO pins.

This routine is used to connect a SPI peripheral to the IO-pads and DMA channel given in the arguments. Depending on the IO-pads requested, the routing is done either using the IO\_mux or using the GPIO matrix.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to be routed
- `bus_config`: Pointer to a `spi_bus_config` struct detailing the GPIO pins
- `dma_chan`: DMA-channel (1 or 2) to use, or 0 for no DMA.
- `flags`: Combination of `SPICOMMON_BUSFLAG_*` flags
- `is_native`: A value of 'true' will be written to this address if the GPIOs can be routed using the IO\_mux, 'false' if the GPIO matrix is used.

`esp_err_t spicommon_bus_free_io` (*spi\_host\_device\_t* host)  
Free the IO used by a SPI peripheral.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to be freed

void `spicommon_cs_initialize` (*spi\_host\_device\_t* host, int cs\_io\_num, int cs\_num, int force\_gpio\_matrix)  
Initialize a Chip Select pin for a specific SPI peripheral.

#### Parameters

- `host`: SPI peripheral
- `cs_io_num`: GPIO pin to route
- `cs_num`: CS id to route
- `force_gpio_matrix`: If true, CS will always be routed through the GPIO matrix. If false, if the GPIO number allows it, the routing will happen through the IO\_mux.

void `spicommon_cs_free` (*spi\_host\_device\_t* host, int cs\_num)  
Free a chip select line.

#### Parameters

- `host`: SPI peripheral
- `cs_num`: CS id to free

void **spicommon\_setup\_dma\_desc\_links** (*lldesc\_t \*dmdesc*, *int len*, **const** *uint8\_t \*data*, *bool isrx*)  
Setup a DMA link chain.

This routine will set up a chain of linked DMA descriptors in the array pointed to by *dmdesc*. Enough DMA descriptors will be used to fit the buffer of *len* bytes in, and the descriptors will point to the corresponding positions in *buffer* and linked together. The end result is that feeding *dmdesc[0]* into DMA hardware results in the entirety *len* bytes of *data* being read or written.

#### Parameters

- *dmdesc*: Pointer to array of DMA descriptors big enough to be able to convey *len* bytes
- *len*: Length of buffer
- *data*: Data buffer to use for DMA transfer
- *isrx*: True if data is to be written into *data*, false if it's to be read from *data*.

*spi\_dev\_t \****spicommon\_hw\_for\_host** (*spi\_host\_device\_t host*)  
Get the position of the hardware registers for a specific SPI host.

**Return** A register descriptor struct pointer, pointed at the hardware registers

#### Parameters

- *host*: The SPI host

void **spicommon\_freeze\_cs** (*spi\_host\_device\_t host*)  
Temporarily connect CS signal input to high to avoid slave detecting unexpected transactions.

**Note** Don't use this in the application.

#### Parameters

- *host*: The spi host.

void **spicommon\_restore\_cs** (*spi\_host\_device\_t host*, *int cs\_io\_num*, *bool iomux*)  
Use this function instead of *cs\_initial* to avoid overwrite the output config This is used in test by internal gpio matrix connections

**Note** Don't use this in the application.

#### Parameters

- *host*: The spi host.
- *cs\_io\_num*: GPIO number of the CS pin.
- *iomux*: The peripheral is using iomux pins.

*int* **spicommon\_irqsource\_for\_host** (*spi\_host\_device\_t host*)  
Get the IRQ source for a specific SPI host.

**Return** The hosts IRQ source

#### Parameters

- *host*: The SPI host

bool **spicommon\_dmaworkaround\_req\_reset** (int *dmachan*, *dmaworkaround\_cb\_t* *cb*, void *\*arg*)  
Request a reset for a certain DMA channel.

Essentially, when a reset is needed, a driver can request this using `spicommon_dmaworkaround_req_reset`. This is supposed to be called with an user-supplied function as an argument. If both DMA channels are idle, this call will reset the DMA subsystem and return true. If the other DMA channel is still busy, it will return false; as soon as the other DMA channel is done, however, it will reset the DMA subsystem and call the callback. The callback is then supposed to be used to continue the SPI drivers activity.

**Note** In some (well-defined) cases in the ESP32 (at least rev v.0 and v.1), a SPI DMA channel will get confused. This can be remedied by resetting the SPI DMA hardware in case this happens. Unfortunately, the reset knob used for this will reset *both* DMA channels, and as such can only be done safely when both DMA channels are idle. These functions coordinate this.

**Return** True when a DMA reset could be executed immediately. False when it could not; in this case the callback will be called with the specified argument when the logic can execute a reset, after that reset.

#### Parameters

- *dmachan*: DMA channel associated with the SPI host that needs a reset
- *cb*: Callback to call in case DMA channel cannot be reset immediately
- *arg*: Argument to the callback

bool **spicommon\_dmaworkaround\_reset\_in\_progress** ()  
Check if a DMA reset is requested but has not completed yet.

**Return** True when a DMA reset is requested but hasn't completed yet. False otherwise.

void **spicommon\_dmaworkaround\_idle** (int *dmachan*)  
Mark a DMA channel as idle.

A call to this function tells the workaround logic that this channel will not be affected by a global SPI DMA reset.

void **spicommon\_dmaworkaround\_transfer\_active** (int *dmachan*)  
Mark a DMA channel as active.

A call to this function tells the workaround logic that this channel will be affected by a global SPI DMA reset, and a reset like that should not be attempted.

## Structures

**struct spi\_bus\_config\_t**

This is a configuration structure for a SPI bus.

You can use this structure to specify the GPIO pins of the bus. Normally, the driver will use the GPIO matrix to route the signals. An exception is made when all signals either can be routed through the IO\_MUX or are -1. In that case, the IO\_MUX is used, allowing for >40MHz speeds.

**Note** Be advised that the slave driver does not use the quadwp/quadhd lines and fields in `spi_bus_config_t` referring to these lines will be ignored and can thus safely be left uninitialized.

## Public Members

**int mosi\_io\_num**  
GPIO pin for Master Out Slave In (=spi\_d) signal, or -1 if not used.

**int miso\_io\_num**  
GPIO pin for Master In Slave Out (=spi\_q) signal, or -1 if not used.

**int sclk\_io\_num**  
GPIO pin for Spi CLoCK signal, or -1 if not used.

**int quadwp\_io\_num**  
GPIO pin for WP (Write Protect) signal which is used as D2 in 4-bit communication modes, or -1 if not used.

**int quadhd\_io\_num**  
GPIO pin for HD (Hold) signal which is used as D3 in 4-bit communication modes, or -1 if not used.

**int max\_transfer\_sz**  
Maximum transfer size, in bytes. Defaults to 4094 if 0.

## Macros

**SPI\_MAX\_DMA\_LEN**

**SPICOMMON\_BUSFLAG\_SLAVE**  
Initialize I/O in slave mode.

**SPICOMMON\_BUSFLAG\_MASTER**  
Initialize I/O in master mode.

**SPICOMMON\_BUSFLAG\_QUAD**  
Also initialize WP/HD pins, if specified.

## Type Definitions

**typedef void (\*dmaworkaround\_cb\_t) (void \*arg)**  
Callback, to be called when a DMA engine reset is completed

## Enumerations

**enum spi\_host\_device\_t**  
Enum with the three SPI peripherals that are software-accessible in it.

*Values:*

**SPI\_HOST =0**  
SPI1, SPI.

**HSPI\_HOST =1**  
SPI2, HSPI.

**VSPI\_HOST =2**  
SPI3, VSPI.

## API Reference - SPI Master

### Header File

- `driver/include/driver/spi_master.h`

### Functions

`esp_err_t spi_bus_initialize` (*spi\_host\_device\_t* host, **const** *spi\_bus\_config\_t* \*bus\_config, int *dma\_chan*)

Initialize a SPI bus.

**Warning** For now, only supports HSPI and VSPI.

**Warning** If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

#### Return

- `ESP_ERR_INVALID_ARG` if configuration is invalid
- `ESP_ERR_INVALID_STATE` if host already is in use
- `ESP_ERR_NO_MEM` if out of memory
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral that controls this bus
- `bus_config`: Pointer to a *spi\_bus\_config\_t* struct specifying how the host should be initialized
- `dma_chan`: Either channel 1 or 2, or 0 in the case when no DMA is required. Selecting a DMA channel for a SPI bus allows transfers on the bus to have sizes only limited by the amount of internal memory. Selecting no DMA channel (by passing the value 0) limits the amount of bytes transferred to a maximum of 32.

`esp_err_t spi_bus_free` (*spi\_host\_device\_t* host)

Free a SPI bus.

**Warning** In order for this to succeed, all devices have to be removed first.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_INVALID_STATE` if not all devices on the bus are freed
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to free

`esp_err_t spi_bus_add_device` (*spi\_host\_device\_t* host, *spi\_device\_interface\_config\_t* \*dev\_config, *spi\_device\_handle\_t* \*handle)

Allocate a device on a SPI bus.

This initializes the internal structures for a device, plus allocates a CS pin on the indicated SPI master peripheral and routes it to the indicated GPIO. All SPI master devices have three CS pins and can thus control up to three devices.

**Note** While in general, speeds up to 80MHz on the dedicated SPI pins and 40MHz on GPIO-matrix-routed pins are supported, full-duplex transfers routed over the GPIO matrix only support speeds up to 26MHz.

**Return**

- ESP\_ERR\_INVALID\_ARG if parameter is invalid
- ESP\_ERR\_NOT\_FOUND if host doesn't have any free CS slots
- ESP\_ERR\_NO\_MEM if out of memory
- ESP\_OK on success

**Parameters**

- host: SPI peripheral to allocate device on
- dev\_config: SPI interface protocol config for the device
- handle: Pointer to variable to hold the device handle

esp\_err\_t **spi\_bus\_remove\_device** (*spi\_device\_handle\_t* handle)  
Remove a device from the SPI bus.

**Return**

- ESP\_ERR\_INVALID\_ARG if parameter is invalid
- ESP\_ERR\_INVALID\_STATE if device already is freed
- ESP\_OK on success

**Parameters**

- handle: Device handle to free

esp\_err\_t **spi\_device\_queue\_trans** (*spi\_device\_handle\_t* handle, *spi\_transaction\_t* \*trans\_desc, Tick-  
Type\_t ticks\_to\_wait)  
Queue a SPI transaction for execution.

**Return**

- ESP\_ERR\_INVALID\_ARG if parameter is invalid
- ESP\_ERR\_TIMEOUT if there was no room in the queue before ticks\_to\_wait expired
- ESP\_ERR\_NO\_MEM if allocating DMA-capable temporary buffer failed
- ESP\_OK on success

**Parameters**

- handle: Device handle obtained using spi\_host\_add\_dev
- trans\_desc: Description of transaction to execute
- ticks\_to\_wait: Ticks to wait until there's room in the queue; use portMAX\_DELAY to never time out.

`esp_err_t spi_device_get_trans_result` (*spi\_device\_handle\_t* handle, *spi\_transaction\_t* *\*\*trans\_desc*, *TickType\_t* ticks\_to\_wait)

Get the result of a SPI transaction queued earlier.

This routine will wait until a transaction to the given device (queued earlier with `spi_device_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.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_TIMEOUT` if there was no completed transaction before `ticks_to_wait` expired
- `ESP_OK` on success

#### Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Pointer to variable able to contain a pointer to the description of the transaction that is executed. The descriptor should not be modified until the descriptor is returned by `spi_device_get_trans_result`.
- `ticks_to_wait`: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

`esp_err_t spi_device_transmit` (*spi\_device\_handle\_t* handle, *spi\_transaction\_t* *\*trans\_desc*)

Do a SPI transaction.

Essentially does the same as `spi_device_queue_trans` followed by `spi_device_get_trans_result`. Do not use this when there is still a transaction queued that hasn't been finalized using `spi_device_get_trans_result`.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

#### Parameters

- `handle`: Device handle obtained using `spi_host_add_dev`
- `trans_desc`: Description of transaction to execute

## Structures

**struct spi\_device\_interface\_config\_t**

This is a configuration for a SPI slave device that is connected to one of the SPI buses.

#### Public Members

`uint8_t command_bits`

Default amount of bits in command phase (0-16), used when `SPI_TRANS_VARIABLE_CMD` is not used, otherwise ignored.

`uint8_t address_bits`

Default amount of bits in address phase (0-64), used when `SPI_TRANS_VARIABLE_ADDR` is not used, otherwise ignored.

`uint8_t dummy_bits`

Amount of dummy bits to insert between address and data phase.

`uint8_t mode`

SPI mode (0-3)

`uint8_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.

`uint8_t cs_ena_pretrans`

Amount of SPI bit-cycles the cs should be activated before the transmission (0-16). This only works on half-duplex transactions.

`uint8_t cs_ena_posttrans`

Amount of SPI bit-cycles the cs should stay active after the transmission (0-16)

`int clock_speed_hz`

Clock speed, in Hz.

`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.

*transaction\_cb\_t post\_cb*

Callback to be called after a transmission has completed. This callback is called within interrupt context.

`struct spi_transaction_t`

This structure describes one SPI transaction. The descriptor should not be modified until the transaction finishes.

## Public Members

`uint32_t flags`

Bitwise OR of `SPI_TRANS_*` flags.

`uint16_t cmd`

Command data, of which the length is set in the `command_bits` of *spi\_device\_interface\_config\_t*.

**NOTE: this field, used to be “command” in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF 3.0.**

- Example: write 0x0123 and `command_bits=12` to send command 0x12, 0x3\_ (in previous version, you may have to write 0x3\_12).

`uint64_t addr`

Address data, of which the length is set in the `address_bits` of *spi\_device\_interface\_config\_t*. **NOTE:**

**this field, used to be “address” in ESP-IDF 2.1 and before, is re-written to be used in a new way in ESP-IDF3.0.**

- Example: write 0x123400 and `address_bits=24` to send address of 0x12, 0x34, 0x00 (in previous version, you may have to write 0x12340000).

**size\_t length**

Total data length, in bits.

**size\_t rxlength**

Total data length received, should be not greater than `length` in full-duplex mode (0 defaults this to the value of `length`).

**void \*user**

User-defined variable. Can be used to store eg transaction ID.

**const void \*tx\_buffer**

Pointer to transmit buffer, or NULL for no MOSI phase.

**uint8\_t tx\_data[4]**

If `SPI_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_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.

## Macros

**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\_TRANS\_MODE\_DIO**

Transmit/receive data in 2-bit mode.

**SPI\_TRANS\_MODE\_QIO**

Transmit/receive data in 4-bit mode.

**SPI\_TRANS\_MODE\_DIOQIO\_ADDR**

Also transmit address in mode selected by SPI\_MODE\_DIO/SPI\_MODE\_QIO.

**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\_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*.

## Type Definitions

```
typedef struct spi_transaction_t spi_transaction_t
```

```
typedef void (*transaction_cb_t) (spi_transaction_t *trans)
```

```
typedef struct spi_device_t *spi_device_handle_t
```

Handle for a device on a SPI bus.

## 2.4.13 SPI Slave driver

### Overview

The ESP32 has four SPI peripheral devices, called SPI0, SPI1, HSPI and VSPI. SPI0 is entirely dedicated to the flash cache the ESP32 uses to map the SPI flash device it is connected to into memory. SPI1 is connected to the same hardware lines as SPI0 and is used to write to the flash chip. HSPI and VSPI are free to use, and with the *spi\_slave* driver, these can be used as a SPI slave, driven from a connected SPI master.

### The *spi\_slave* driver

The *spi\_slave* driver allows using the HSPI and/or VSPI peripheral as a full-duplex SPI slave. It can make use of DMA to send/receive transactions of arbitrary length.

### Terminology

The *spi\_slave* driver uses the following terms:

- Host: The SPI peripheral inside the ESP32 initiating the SPI transmissions. One of HSPI or VSPI.
- Bus: The SPI bus, common to all SPI devices connected to a master. In general the bus consists of the miso, mosi, sclk and optionally quadwp and quadhd signals. The SPI slaves are connected to these signals in parallel. Each SPI slave is also connected to one CS signal.

- `miso` - Also known as `q`, this is the output of the serial stream from the ESP32 to the SPI master
  - `mosi` - Also known as `d`, this is the output of the serial stream from the SPI master to the ESP32
  - `clk` - Clock signal. Each data bit is clocked out or in on the positive or negative edge of this signal
  - `cs` - Chip Select. An active Chip Select delineates a single transaction to/from a slave.
- Transaction: One instance of CS going active, data transfer from and to a master happening, and CS going inactive again. Transactions are atomic, as in they will never be interrupted by another transaction.

## SPI transactions

A full-duplex SPI transaction starts with the master pulling CS low. After this happens, the master starts sending out clock pulses on the CLK line: every clock pulse causes a data bit to be shifted from the master to the slave on the MOSI line and vice versa on the MISO line. At the end of the transaction, the master makes CS high again.

## Using the `spi_slave` driver

- Initialize a SPI peripheral as a slave by calling `spi_slave_initialize`. Make sure to set the correct IO pins in the `bus_config` struct. Take care to set signals that are not needed to -1. A DMA channel (either 1 or 2) must be given if transactions will be larger than 32 bytes, if not the `dma_chan` parameter may be 0.
- To set up a transaction, fill one or more `spi_transaction_t` structure with any transaction parameters you need. Either queue all transactions by calling `spi_slave_queue_trans`, later querying the result using `spi_slave_get_trans_result`, or handle all requests synchronously by feeding them into `spi_slave_transmit`. The latter two functions will block until the master 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, data to be transferred to or from a device will be read from or written to a chunk of memory indicated by the `rx_buffer` and `tx_buffer` members of the transaction structure. The SPI driver may decide to use DMA for transfers, so these buffers should be allocated in DMA-capable memory using `pvPortMallocCaps(size, MALLOC_CAP_DMA)`.

The amount of data written to the buffers is limited by the `length` member of the transaction structure: the driver will never read/write more data than indicated there. The `length` cannot define the actual length of the SPI transaction; this is determined by the master as it drives the clock and CS lines. The actual length transferred can be read from the `trans_len` member of the `spi_slave_transaction_t` structure after transaction. In case the length of the transmission is larger than the buffer length, only the start of the transmission will be sent and received, and the `trans_len` is set to `length` instead of the actual length. It's recommended to set `length` longer than the maximum length expected if the `trans_len` is required. In case the transmission length is shorter than the buffer length, only data up to the length of the buffer will be exchanged.

Warning: Due to a design peculiarity in the ESP32, if the amount of bytes sent by the master or the length of the transmission queues in the slave driver, in bytes, is not both larger than eight and dividable by four, the SPI hardware can fail to write the last one to seven bytes to the receive buffer.

## Application Example

Slave/master communication: [peripherals/spi\\_slave](#).

## API Reference

### Header File

- `driver/include/driver/spi_slave.h`

### Functions

`esp_err_t spi_slave_initialize` (*spi\_host\_device\_t* host, **const** *spi\_bus\_config\_t* \*bus\_config, **const** *spi\_slave\_interface\_config\_t* \*slave\_config, int dma\_chan)

Initialize a SPI bus as a slave interface.

**Warning** For now, only supports HSPI and VSPI.

**Warning** If a DMA channel is selected, any transmit and receive buffer used should be allocated in DMA-capable memory.

#### Return

- `ESP_ERR_INVALID_ARG` if configuration is invalid
- `ESP_ERR_INVALID_STATE` if host already is in use
- `ESP_ERR_NO_MEM` if out of memory
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to use as a SPI slave interface
- `bus_config`: Pointer to a *spi\_bus\_config\_t* struct specifying how the host should be initialized
- `slave_config`: Pointer to a *spi\_slave\_interface\_config\_t* struct specifying the details for the slave interface
- `dma_chan`: Either 1 or 2. A SPI bus used by this driver must have a DMA channel associated with it. The SPI hardware has two DMA channels to share. This parameter indicates which one to use.

`esp_err_t spi_slave_free` (*spi\_host\_device\_t* host)

Free a SPI bus claimed as a SPI slave interface.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_ERR_INVALID_STATE` if not all devices on the bus are freed
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to free

`esp_err_t spi_slave_queue_trans` (*spi\_host\_device\_t* host, **const** *spi\_slave\_transaction\_t* \*trans\_desc, TickType\_t ticks\_to\_wait)

Queue a SPI transaction for execution.

Queues a SPI transaction to be executed by this slave device. (The transaction queue size was specified when the slave device was initialised via `spi_slave_initialize`.) This function may block if the queue is full (depending on

the `ticks_to_wait` parameter). No SPI operation is directly initiated by this function, the next queued transaction will happen when the master initiates a SPI transaction by pulling down CS and sending out clock signals.

This function hands over ownership of the buffers in `trans_desc` to the SPI slave driver; the application is not to access this memory until `spi_slave_queue_trans` is called to hand ownership back to the application.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral that is acting as a slave
- `trans_desc`: Description of transaction to execute. Not const because we may want to write status back into the transaction description.
- `ticks_to_wait`: Ticks to wait until there's room in the queue; use `portMAX_DELAY` to never time out.

```
esp_err_t spi_slave_get_trans_result(spi_host_device_t host, spi_slave_transaction_t
                                   **trans_desc, TickType_t ticks_to_wait)
```

Get the result of a SPI transaction queued earlier.

This routine will wait until a transaction to the given device (queued earlier with `spi_slave_queue_trans`) has successfully completed. It will then return the description of the completed transaction so software can inspect the result and e.g. free the memory or re-use the buffers.

It is mandatory to eventually use this function for any transaction queued by `spi_slave_queue_trans`.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to that is acting as a slave
- `trans_desc`: Pointer to variable able to contain a pointer to the description of the transaction that is executed
- `ticks_to_wait`: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

```
esp_err_t spi_slave_transmit(spi_host_device_t host, spi_slave_transaction_t *trans_desc, TickType_t
                             ticks_to_wait)
```

Do a SPI transaction.

Essentially does the same as `spi_slave_queue_trans` followed by `spi_slave_get_trans_result`. Do not use this when there is still a transaction queued that hasn't been finalized using `spi_slave_get_trans_result`.

#### Return

- `ESP_ERR_INVALID_ARG` if parameter is invalid
- `ESP_OK` on success

#### Parameters

- `host`: SPI peripheral to that is acting as a slave

- `trans_desc`: Pointer to variable able to contain a pointer to the description of the transaction that is executed. Not const because we may want to write status back into the transaction description.
- `ticks_to_wait`: Ticks to wait until there's a returned item; use `portMAX_DELAY` to never time out.

## Structures

### **struct spi\_slave\_interface\_config\_t**

This is a configuration for a SPI host acting as a slave device.

#### Public Members

int **spics\_io\_num**

CS GPIO pin for this device.

uint32\_t **flags**

Bitwise OR of `SPI_SLAVE_*` flags.

int **queue\_size**

Transaction queue size. This sets how many transactions can be 'in the air' (queued using `spi_slave_queue_trans` but not yet finished using `spi_slave_get_trans_result`) at the same time.

uint8\_t **mode**

SPI mode (0-3)

*slave\_transaction\_cb\_t* **post\_setup\_cb**

Callback called after the SPI registers are loaded with new data.

*slave\_transaction\_cb\_t* **post\_trans\_cb**

Callback called after a transaction is done.

### **struct spi\_slave\_transaction\_t**

This structure describes one SPI transaction

#### Public Members

size\_t **length**

Total data length, in bits.

size\_t **trans\_len**

Transaction data length, in bits.

const void **\*tx\_buffer**

Pointer to transmit buffer, or NULL for no MOSI phase.

void **\*rx\_buffer**

Pointer to receive buffer, or NULL for no MISO phase.

void **\*user**

User-defined variable. Can be used to store eg transaction ID.

## Macros

### **SPI\_SLAVE\_TXBIT\_LSBFIRST**

Transmit command/address/data LSB first instead of the default MSB first.

**SPI\_SLAVE\_RXBIT\_LSBFIRST**

Receive data LSB first instead of the default MSB first.

**SPI\_SLAVE\_BIT\_LSBFIRST**

Transmit and receive LSB first.

## Type Definitions

```
typedef struct spi_slave_transaction_t spi_slave_transaction_t
```

```
typedef void (*slave_transaction_cb_t) (spi_slave_transaction_t *trans)
```

## 2.4.14 TIMER

### Introduction

The ESP32 chip contains two hardware timer groups. Each group has two general-purpose hardware timers. They are all 64-bit generic timers based on 16-bit prescalers and 64-bit auto-reload-capable up / down counters.

### Functional Overview

Typical steps to configure an operate the timer are described in the following sections:

- *Timer Initialization* - what parameters should be set up to get the timer working and what specific functionality is provided depending on the set up.
- *Timer Control* - how to read the timer's value, pause / start the timer, and change how it operates.
- *Alarms* - setting and using alarms.
- *Interrupts* - how to enable and use interrupts.

### Timer Initialization

The two timer groups on-board of the ESP32 are identified using `timer_group_t`. Individual timers in a group are identified with `timer_idx_t`. The two groups, each having two timers, provide the total of four individual timers to our disposal.

Before starting the timer, it should be initialized by calling `timer_init()`. This function should be provided with a structure `timer_config_t` to define how timer should operate. In particular the following timer's parameters may be set:

- **Divider:** How quickly the timer's counter is "ticking". This depends on the setting of `divider`, that will be used as divisor of the incoming 80 MHz APB\_CLK clock.
- **Mode:** If the the counter is incrementing or decrementing, defined using `counter_dir` by selecting one of values from `timer_count_dir_t`.
- **Counter Enable:** If the counter is enabled, then it will start incrementing / decrementing immediately after calling `timer_init()`. This action is set using `counter_en` by selecting one of vales from `timer_start_t`.
- **Alarm Enable:** Determined by the setting of `alarm_en`.
- **Auto Reload:** Whether the counter should `auto_reload` a specific initial value on the timer's alarm, or continue incrementing or decrementing.

- **Interrupt Type:** Whether an interrupt is triggered on timer's alarm. Set the value defined in `timer_intr_mode_t`.

To get the current values of the timers settings, use function `timer_get_config()`.

## Timer Control

Once the timer is configured and enabled, it is already “ticking”. To check it's current value call `timer_get_counter_value()` or `timer_get_counter_time_sec()`. To set the timer to specific starting value call `timer_set_counter_value()`.

The timer may be paused at any time by calling `timer_pause()`. To start it again call `timer_start()`.

To change how the timer operates you can call once more `timer_init()` described in section *Timer Initialization*. Another option is to use dedicated functions to change individual settings:

- **Divider** value - `timer_set_divider()`. **Note:** the timer should be paused when changing the divider to avoid unpredictable results. If the timer is already running, `timer_set_divider()` will first pause the timer, change the divider, and finally start the timer again.
- **Mode** (whether the counter incrementing or decrementing) - `timer_set_counter_mode()`
- **Auto Reload** counter on alarm - `timer_set_auto_reload()`

## Alarms

To set an alarm, call function `timer_set_alarm_value()` and then enable it with `timer_set_alarm()`. The alarm may be also enabled during the timer initialization stage, when `timer_init()` is called.

After the alarm is enabled and the timer reaches the alarm value, depending on configuration, the following two actions may happen:

- An interrupt will be triggered, if previously configured. See section *Interrupts* how to configure interrupts.
- When `auto_reload` is enabled, the timer's counter will be reloaded to start counting from specific initial value. The value to start should be set in advance with `timer_set_counter_value()`.

---

**Note:** The alarm will be triggered immediately, if an alarm value is set and the timer has already passed this value.

---

To check what alarm value has been set up, call `timer_get_alarm_value()`.

## Interrupts

Registration of the interrupt handler for a specific timer group and timer is done by calling `timer_isr_register()`.

To enable interrupts for a timer group call `timer_group_intr_enable()`. To do it for a specific timer, call `timer_enable_intr()`. Disabling of interrupts is done with corresponding functions `timer_group_intr_disable()` and `timer_disable_intr()`.

When servicing an interrupt within an ISR, the interrupt need to explicitly cleared. To do so, set the `TIMERGN.int_clr_timers.tM` structure defined in `soc/esp32/include/soc/timer_group_struct.h`, where N is the timer group number [0, 1] and M is the timer number [0, 1]. For example to clear an interrupt for the timer 1 in the timer group 0, call the following:

```
TIMERG0.int_clr_timers.t1 = 1
```

See the application example below how to use interrupts.

## Application Example

The 64-bit hardware timer example: [peripherals/timer\\_group](#).

## API Reference

### Header File

- [driver/include/driver/timer.h](#)

### Functions

esp\_err\_t **timer\_get\_counter\_value**(*timer\_group\_t* group\_num, *timer\_idx\_t* timer\_num, uint64\_t \*timer\_val)

Read the counter value of hardware timer.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- group\_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer\_num: Timer index, 0 for hw\_timer[0] & 1 for hw\_timer[1]
- timer\_val: Pointer to accept timer counter value.

esp\_err\_t **timer\_get\_counter\_time\_sec**(*timer\_group\_t* group\_num, *timer\_idx\_t* timer\_num, double \*time)

Read the counter value of hardware timer, in unit of a given scale.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- group\_num: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- timer\_num: Timer index, 0 for hw\_timer[0] & 1 for hw\_timer[1]
- time: Pointer, type of double\*, to accept timer counter value, in seconds.

esp\_err\_t **timer\_set\_counter\_value**(*timer\_group\_t* group\_num, *timer\_idx\_t* timer\_num, uint64\_t load\_val)

Set counter value to hardware timer.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `load_val`: Counter value to write to the hardware timer.

`esp_err_t timer_start` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`)  
Start the counter of hardware timer.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`

`esp_err_t timer_pause` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`)  
Pause the counter of hardware timer.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`

`esp_err_t timer_set_counter_mode` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`,  
*timer\_count\_dir\_t* `counter_dir`)  
Set counting mode for hardware timer.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `counter_dir`: Counting direction of timer, count-up or count-down

`esp_err_t timer_set_auto_reload` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`,  
*timer\_autoreload\_t* `reload`)  
Enable or disable counter reload function when alarm event occurs.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `reload`: Counter reload mode.

`esp_err_t timer_set_divider` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`, `uint32_t` `divider`)  
 Set hardware timer source clock divider. Timer groups clock are divider from APB clock.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `divider`: Timer clock divider value. The divider's range is from from 2 to 65536.

`esp_err_t timer_set_alarm_value` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`, `uint64_t` `alarm_value`)

Set timer alarm value.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_value`: A 64-bit value to set the alarm value.

`esp_err_t timer_get_alarm_value` (*timer\_group\_t* `group_num`, *timer\_idx\_t* `timer_num`, `uint64_t` `*alarm_value`)

Get timer alarm value.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- `group_num`: Timer group, 0 for TIMERG0 or 1 for TIMERG1
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_value`: Pointer of A 64-bit value to accept the alarm value.

`esp_err_t timer_set_alarm(timer_group_t group_num, timer_idx_t timer_num, timer_alarm_t alarm_en)`

Get timer alarm value.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `group_num`: Timer group, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `alarm_en`: To enable or disable timer alarm function.

`esp_err_t timer_isr_register(timer_group_t group_num, timer_idx_t timer_num, void (*fn)) void *, void *arg, int intr_alloc_flags, timer_isr_handle_t *handle` Register Timer interrupt handler, the handler is an ISR. The handler will be attached to the same CPU core that this function is running on.

**Note** If the `intr_alloc_flags` value `ESP_INTR_FLAG_IRAM` is set, the handler function must be declared with `IRAM_ATTR` attribute and can only call functions in IRAM or ROM. It cannot call other timer APIs. Use direct register access to configure timers from inside the ISR in this case.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `group_num`: Timer group number
- `timer_num`: Timer index of timer group
- `fn`: Interrupt handler function.
- `arg`: Parameter for handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info.
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

`esp_err_t timer_init(timer_group_t group_num, timer_idx_t timer_num, const timer_config_t *config)` Initializes and configure the timer.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index, 0 for `hw_timer[0]` & 1 for `hw_timer[1]`
- `config`: Pointer to timer initialization parameters.

`esp_err_t timer_get_config(timer_group_t group_num, timer_idx_t timer_num, timer_config_t *config)`  
Get timer configure value.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- group\_num: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- timer\_num: Timer index, 0 for hw\_timer[0] & 1 for hw\_timer[1]
- config: Pointer of struct to accept timer parameters.

`esp_err_t timer_group_intr_enable(timer_group_t group_num, uint32_t en_mask)`  
Enable timer group interrupt, by enable mask.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- group\_num: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- en\_mask: Timer interrupt enable mask. Use TIMG\_T0\_INT\_ENA\_M to enable t0 interrupt Use TIMG\_T1\_INT\_ENA\_M to enable t1 interrupt

`esp_err_t timer_group_intr_disable(timer_group_t group_num, uint32_t disable_mask)`  
Disable timer group interrupt, by disable mask.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- group\_num: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- disable\_mask: Timer interrupt disable mask. Use TIMG\_T0\_INT\_ENA\_M to disable t0 interrupt Use TIMG\_T1\_INT\_ENA\_M to disable t1 interrupt

`esp_err_t timer_enable_intr(timer_group_t group_num, timer_idx_t timer_num)`  
Enable timer interrupt.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

**Parameters**

- group\_num: Timer group number, 0 for TIMERG0 or 1 for TIMERG1
- timer\_num: Timer index.

`esp_err_t timer_disable_intr(timer_group_t group_num, timer_idx_t timer_num)`  
Disable timer interrupt.

#### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Parameter error

#### Parameters

- `group_num`: Timer group number, 0 for `TIMERG0` or 1 for `TIMERG1`
- `timer_num`: Timer index.

## Structures

`struct timer_config_t`

Data structure with timer's configuration settings.

#### Public Members

bool `alarm_en`

Timer alarm enable

bool `counter_en`

Counter enable

`timer_intr_mode_t intr_type`

Interrupt mode

`timer_count_dir_t counter_dir`

Counter direction

bool `auto_reload`

Timer auto-reload

uint32\_t `divider`

Counter clock divider. The divider's range is from from 2 to 65536.

## Macros

`TIMER_BASE_CLK`

Frequency of the clock on the input of the timer groups

## Type Definitions

`typedef intr_handle_t timer_isr_handle_t`

Interrupt handle, used in order to free the isr after use. Aliases to an int handle for now.

## Enumerations

### **enum timer\_group\_t**

Selects a Timer-Group out of 2 available groups.

*Values:*

**TIMER\_GROUP\_0** = 0

Hw timer group 0

**TIMER\_GROUP\_1** = 1

Hw timer group 1

**TIMER\_GROUP\_MAX**

### **enum timer\_idx\_t**

Select a hardware timer from timer groups.

*Values:*

**TIMER\_0** = 0

Select timer0 of GROUPx

**TIMER\_1** = 1

Select timer1 of GROUPx

**TIMER\_MAX**

### **enum timer\_count\_dir\_t**

Decides the direction of counter.

*Values:*

**TIMER\_COUNT\_DOWN** = 0

Descending Count from cnt.highcnt.low

**TIMER\_COUNT\_UP** = 1

Ascending Count from Zero

**TIMER\_COUNT\_MAX**

### **enum timer\_start\_t**

Decides whether timer is on or paused.

*Values:*

**TIMER\_PAUSE** = 0

Pause timer counter

**TIMER\_START** = 1

Start timer counter

### **enum timer\_alarm\_t**

Decides whether to enable alarm mode.

*Values:*

**TIMER\_ALARM\_DIS** = 0

Disable timer alarm

**TIMER\_ALARM\_EN** = 1

Enable timer alarm

**TIMER\_ALARM\_MAX**

**enum timer\_intr\_mode\_t**

Select interrupt type if running in alarm mode.

*Values:*

**TIMER\_INTR\_LEVEL = 0**

Interrupt mode: level mode

**TIMER\_INTR\_MAX**

**enum timer\_autoreload\_t**

Select if Alarm needs to be loaded by software or automatically reload by hardware.

*Values:*

**TIMER\_AUTORELOAD\_DIS = 0**

Disable auto-reload: hardware will not load counter value after an alarm event

**TIMER\_AUTORELOAD\_EN = 1**

Enable auto-reload: hardware will load counter value after an alarm event

**TIMER\_AUTORELOAD\_MAX**

## 2.4.15 Touch Sensor

### Introduction

A touch-sensor system is built on a substrate which carries electrodes and relevant connections under a protective flat surface. When a user touches the surface, the capacitance variation is triggered and a binary signal is generated to indicate whether the touch is valid.

ESP32 can provide up to 10 capacitive touch pads / GPIOs. The sensing pads can be arranged in different combinations (e.g. matrix, slider), so that a larger area or more points can be detected. The touch pad sensing process is under the control of a hardware-implemented finite-state machine (FSM) which is initiated by software or a dedicated hardware timer.

Design, operation and control registers of touch sensor are discussed in [ESP32 Technical Reference Manual \(PDF\)](#). Please refer to it for additional details how this subsystem works.

### Functionality Overview

Description of API is broken down into groups of functions to provide quick overview of features like:

- 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 particular function please go to section [API Reference](#). Practical implementation of this API is covered in section [Application Examples](#).

## Initialization

Touch pad driver should be initialized before use by calling function `touch_pad_init()`. This function sets several `._DEFAULT` driver parameters listed in *API Reference* under “Macros”. It also clears information what pads have been touched before (if any) and disables interrupts.

If not required anymore, driver can be disabled by calling `touch_pad_deinit()`.

## Configuration

Enabling of touch sensor functionality for particular GPIO is done with `touch_pad_config()`.

The function `touch_pad_set_fsm_mode()` is used to select whether touch pad measurement (operated by FSM) is started automatically by hardware timer, or by software. If software mode is selected, then use `touch_pad_sw_start()` to start of the FSM.

## Touch State Measurements

The following two functions come handy to read raw or filtered measurements from the sensor:

- `touch_pad_read()`
- `touch_pad_read_filtered()`

They may be used to characterize 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 the touch threshold.

---

**Note:** Start and configure filter before using `touch_pad_read_filtered()` by calling specific filter functions described down below.

---

To see how to use both read functions check `peripherals/touch_pad_read` application example.

## Optimization of Measurements

Touch sensor has several configurable parameters to match characteristics of particular touch pad design. For instance, to sense smaller capacity changes, it is possible to narrow the reference voltage range within which the touch pads are charged / discharged. The high and low reference voltages are set using function `touch_pad_set_voltage()`. A positive side effect, besides ability to discern smaller capacity changes, will be reduction of power consumption for low power applications. A likely negative effect will be increase of measurement noise. If dynamic range of obtained readings is still satisfactory, then further reduction of power consumption may be done by lowering the measurement time with `touch_pad_set_meas_time()`.

The following summarizes available measurement parameters and corresponding ‘set’ functions:

- Touch pad charge / discharge parameters:
  - voltage range: `touch_pad_set_voltage()`
  - speed (slope): `touch_pad_set_cnt_mode()`
- Measure time: `touch_pad_set_meas_time()`

Relationship between voltage range (high / low reference voltages), speed (slope) and measure time is shown on figure below.

The last chart “Output” represents the touch sensor reading, i.e. the count of pulses collected within measure time.

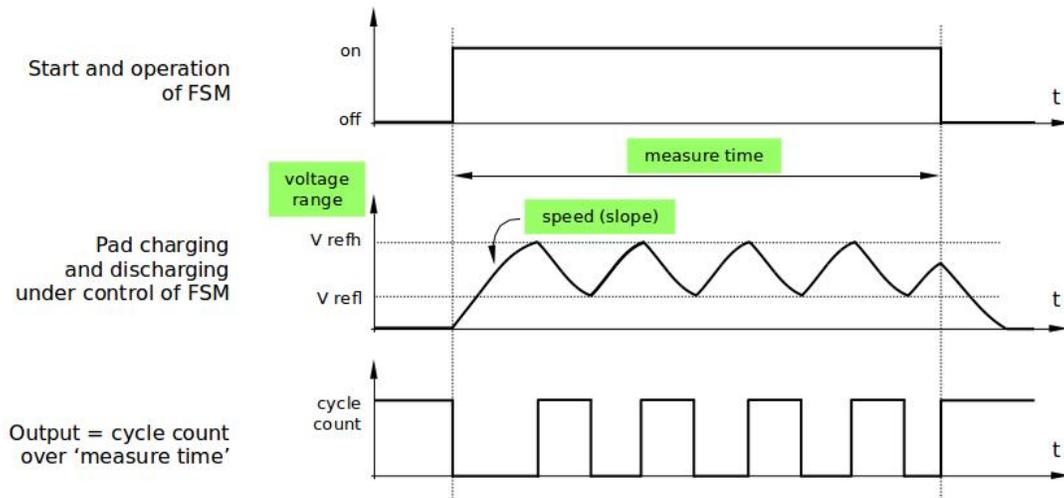


Fig. 9: Touch Pad - relationship between measurement parameters

All functions are provided in pairs to 'set' specific parameter and to 'get' the current parameter's value, e.g. `touch_pad_set_voltage()` and `touch_pad_get_voltage()`.

### Filtering of Measurements

If measurements are noisy, you may filter them with provided API. The filter should be started before first use by calling `touch_pad_filter_start()`.

The filter type is IIR (Infinite Impulse Response) and it has configurable period that can be set with function `touch_pad_set_filter_period()`.

You can stop the filter with `touch_pad_filter_stop()`. If not required anymore, the filter may be deleted by invoking `touch_pad_filter_delete()`.

### Touch Detection

Touch detection is implemented in ESP32's hardware basing on user configured threshold and raw measurements executed by FSM. Use function `touch_pad_get_status()` to check what pads have been touched and `touch_pad_clear_status()` to clear the touch status information.

Hardware touch detection may be also wired to interrupts and this is described in next section.

If measurements are noisy and capacity changes small, then hardware touch detection may be not reliable. To resolve this issue, instead of using hardware detection / provided interrupts, implement measurement filtering and perform touch detection in your own application. See [peripherals/touch\\_pad\\_interrupt](#) for sample implementation of both methods of touch detection.

## Touch Triggered Interrupts

Before enabling an interrupt on touch detection, user should establish touch detection threshold. Use functions described above to read and display sensor measurements when pad is touched and released. Apply a filter when measurements are noisy and relative changes are small. Depending on your application and environmental conditions, test the influence of temperature and power supply voltage changes on measured values.

Once detection threshold is established, it may be set on initialization with `touch_pad_config()` or at the runtime with `touch_pad_set_thresh()`.

In next step configure how interrupts are triggered. They may be triggered below or above threshold and this is set with function `touch_pad_set_trigger_mode()`.

Finally configure and manage interrupt calls using the following functions:

- `touch_pad_isr_register() / touch_pad_isr_deregister()`
- `touch_pad_intr_enable() / touch_pad_intr_disable()`

When interrupts are operational, you can obtain information what particular pad triggered interrupt by invoking `touch_pad_get_status()` and clear pad status with `touch_pad_clear_status()`.

---

**Note:** Interrupts on touch detection operate on raw / unfiltered measurements checked against user established threshold and are implemented in hardware. Enabling software filtering API (see *Filtering of Measurements*) does not affect this process.

---

## Wakeup from Sleep Mode

If touch pad interrupts are used to wakeup the chip from a sleep mode, then user can select certain configuration of pads (SET1 or both SET1 and SET2), that should be touched to trigger the interrupt and cause subsequent wakeup. To do so, use function `touch_pad_set_trigger_source()`.

Configuration of required bit patterns of pads may be managed for each 'SET' with:

- `touch_pad_set_group_mask() / touch_pad_get_group_mask()`
- `touch_pad_clear_group_mask()`

## Application Examples

- Touch sensor read example: [peripherals/touch\\_pad\\_read](#).
- Touch sensor interrupt example: [peripherals/touch\\_pad\\_interrupt](#).

## API Reference

### Header File

- `driver/include/driver/touch_pad.h`

## Functions

`esp_err_t touch_pad_init ()`  
Initialize touch module.

### Return

- ESP\_OK Success
- ESP\_FAIL Touch pad init error

`esp_err_t touch_pad_deinit ()`  
Un-install touch pad driver.

### Return

- ESP\_OK Success
- ESP\_FAIL Touch pad driver not initialized

`esp_err_t touch_pad_config (touch_pad_t touch_num, uint16_t threshold)`  
Configure touch pad interrupt threshold.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG if argument wrong
- ESP\_FAIL if touch pad not initialized

### Parameters

- touch\_num: touch pad index
- threshold: interrupt threshold,

`esp_err_t touch_pad_read (touch_pad_t touch_num, uint16_t *touch_value)`  
get touch sensor counter value. Each touch sensor has a counter to count the number of charge/discharge cycles. When the pad is not 'touched', we can get a number of the counter. When the pad is 'touched', the value in counter will get smaller because of the larger equivalent capacitance. User can use this function to determine the interrupt trigger threshold.

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Touch pad error
- ESP\_FAIL Touch pad not initialized

### Parameters

- touch\_num: touch pad index
- touch\_value: pointer to accept touch sensor value

`esp_err_t touch_pad_read_filtered (touch_pad_t touch_num, uint16_t *touch_value)`  
get filtered touch sensor counter value by IIR filter.

**Note** touch\_pad\_filter\_start has to be called before calling touch\_pad\_read\_filtered. This function can be called from ISR

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG Touch pad error
- ESP\_FAIL Touch pad not initialized

**Parameters**

- touch\_num: touch pad index
- touch\_value: pointer to accept touch sensor value

`esp_err_t touch_pad_isr_handler_register` (void (\*fn)) void \*  
 , void \*arg, int unused, *intr\_handler\_t* \*handle\_unused Register touch-pad ISR,.

**Note** Deprecated function, users should replace this with `touch_pad_isr_register`, because RTC modules share a same interrupt index.

**Return**

- ESP\_OK Success ;
- ESP\_ERR\_INVALID\_ARG GPIO error

**Parameters**

- fn: Pointer to ISR handler
- arg: Parameter for ISR
- unused: Reserved, not used
- handle\_unused: Reserved, not used

`esp_err_t touch_pad_isr_register` (*intr\_handler\_t* fn, void \*arg)  
 Register touch-pad ISR. The handler will be attached to the same CPU core that this function is running on.

**Return**

- ESP\_OK Success ;
- ESP\_ERR\_INVALID\_ARG GPIO error

**Parameters**

- fn: Pointer to ISR handler
- arg: Parameter for ISR

`esp_err_t touch_pad_isr_deregister` (void (\*fn)) void \*  
 , void \*arg Deregister the handler previously registered using `touch_pad_isr_handler_register`.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if a handler matching both fn and arg isn't registered

**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`)

`esp_err_t touch_pad_set_meas_time` (`uint16_t sleep_cycle`, `uint16_t meas_cycle`)  
Set touch sensor measurement and sleep time.

**Return**

- ESP\_OK on success

**Parameters**

- `sleep_cycle`: The touch sensor will sleep after each measurement. `sleep_cycle` decide the interval between each measurement.  $t_{sleep} = sleep\_cycle / (RTC\_SLOW\_CLK \text{ frequency})$ . The approximate frequency value of `RTC_SLOW_CLK` can be obtained using `rtc_clk_slow_freq_get_hz` function.
- `meas_cycle`: The duration of the touch sensor measurement.  $t_{meas} = meas\_cycle / 8M$ , the maximum measure time is  $0xffff / 8M = 8.19 \text{ ms}$

`esp_err_t touch_pad_get_meas_time` (`uint16_t *sleep_cycle`, `uint16_t *meas_cycle`)  
Get touch sensor measurement and sleep time.

**Return**

- ESP\_OK on success

**Parameters**

- `sleep_cycle`: Pointer to accept sleep cycle number
- `meas_cycle`: Pointer to accept measurement cycle count.

`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 reference voltage, if the voltage gap between high and low reference voltage get less, the charging and discharging time would be faster, accordingly, the counter value would be larger. In the case of detecting very slight change of capacitance, we can narrow down the gap so as to increase the sensitivity. On the other hand, narrow voltage gap would also introduce more noise, but we can use a software filter to pre-process the counter value.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if argument is wrong

**Parameters**

- `refh`: the value of DREFH
- `refl`: the value of DREFL
- `atten`: the attenuation on DREFH

`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,.

**Return**

- ESP\_OK on success

**Parameters**

- `refh`: pointer to accept DREFH value

- `refl`: pointer to accept DREFL value
- `atten`: pointer to accept the attenuation on DREFH

`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, the counter value would be small. If the slope is set 7, which is the maximum value, the charging and discharging would be fast, accordingly, the counter value would be larger.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

#### Parameters

- `touch_num`: touch pad index
- `slope`: touch pad charge/discharge speed
- `opt`: the initial voltage

`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.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

#### Parameters

- `touch_num`: touch pad index
- `slope`: pointer to accept touch pad charge/discharge slope
- `opt`: pointer to accept the initial voltage

`esp_err_t touch_pad_io_init` (*touch\_pad\_t touch\_num*)

Initialize touch pad GPIO.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

#### Parameters

- `touch_num`: touch pad index

`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.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

### Parameters

- mode: FSM mode

esp\_err\_t **touch\_pad\_get\_fsm\_mode** (*touch\_fsm\_mode\_t \*mode*)  
Get touch sensor FSM mode.

### Return

- ESP\_OK on success

### Parameters

- mode: pointer to accept FSM mode

esp\_err\_t **touch\_pad\_sw\_start** ()  
Trigger a touch sensor measurement, only support in SW mode of FSM.

### Return

- ESP\_OK on success

esp\_err\_t **touch\_pad\_set\_thresh** (*touch\_pad\_t touch\_num, uint16\_t threshold*)  
Set touch sensor interrupt threshold.

### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if argument is wrong

### Parameters

- touch\_num: touch pad index
- threshold: threshold of touchpad count, refer to touch\_pad\_set\_trigger\_mode to see how to set trigger mode.

esp\_err\_t **touch\_pad\_get\_thresh** (*touch\_pad\_t touch\_num, uint16\_t \*threshold*)  
Get touch sensor interrupt threshold.

### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if argument is wrong

### Parameters

- touch\_num: touch pad index
- threshold: pointer to accept threshold

esp\_err\_t **touch\_pad\_set\_trigger\_mode** (*touch\_trigger\_mode\_t mode*)  
Set touch sensor interrupt trigger mode. Interrupt can be triggered either when counter result is less than threshold or when counter result is more than threshold.

### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if argument is wrong

**Parameters**

- `mode`: touch sensor interrupt trigger mode

`esp_err_t touch_pad_get_trigger_mode(touch_trigger_mode_t *mode)`  
Get touch sensor interrupt trigger mode.

**Return**

- `ESP_OK` on success

**Parameters**

- `mode`: pointer to accept touch sensor interrupt trigger mode

`esp_err_t touch_pad_set_trigger_source(touch_trigger_src_t src)`  
Set touch sensor interrupt trigger source. There are two sets of touch signals. Set1 and set2 can be mapped to several touch signals. Either set will be triggered if at least one of its touch signal is 'touched'. The interrupt can be configured to be generated if set1 is triggered, or only if both sets are triggered.

**Return**

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

**Parameters**

- `src`: touch sensor interrupt trigger source

`esp_err_t touch_pad_get_trigger_source(touch_trigger_src_t *src)`  
Get touch sensor interrupt trigger source.

**Return**

- `ESP_OK` on success

**Parameters**

- `src`: pointer to accept touch sensor interrupt trigger source

`esp_err_t touch_pad_set_group_mask(uint16_t set1_mask, uint16_t set2_mask, uint16_t en_mask)`  
Set touch sensor group mask. Touch pad module has two sets of signals, 'Touched' signal is triggered only if at least one of touch pad in this group is "touched". This function will set the register bits according to the given bitmask.

**Return**

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if argument is wrong

**Parameters**

- `set1_mask`: bitmask of touch sensor signal group1, it's a 10-bit value
- `set2_mask`: bitmask of touch sensor signal group2, it's a 10-bit value
- `en_mask`: bitmask of touch sensor work enable, it's a 10-bit value

`esp_err_t touch_pad_get_group_mask(uint16_t *set1_mask, uint16_t *set2_mask, uint16_t *en_mask)`  
Get touch sensor group mask.

**Return**

- ESP\_OK on success

**Parameters**

- `set1_mask`: pointer to accept bitmask of touch sensor signal group1, it's a 10-bit value
- `set2_mask`: pointer to accept bitmask of touch sensor signal group2, it's a 10-bit value
- `en_mask`: pointer to accept bitmask of touch sensor work enable, it's a 10-bit value

`esp_err_t touch_pad_clear_group_mask (uint16_t set1_mask, uint16_t set2_mask, uint16_t en_mask)`

Clear touch sensor group mask. Touch pad module has two sets of signals, Interrupt is triggered only if at least one of touch pad in this group is "touched". This function will clear the register bits according to the given bitmask.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if argument is wrong

**Parameters**

- `set1_mask`: bitmask touch sensor signal group1, it's a 10-bit value
- `set2_mask`: bitmask touch sensor signal group2, it's a 10-bit value
- `en_mask`: bitmask of touch sensor work enable, it's a 10-bit value

`esp_err_t touch_pad_clear_status ()`

To clear the touch status register, usually use this function in touch ISR to clear status.

**Return**

- ESP\_OK on success

`uint32_t touch_pad_get_status ()`

Get the touch sensor status, usually used in ISR to decide which pads are 'touched'.

**Return**

- touch status

`esp_err_t touch_pad_intr_enable ()`

To enable touch pad interrupt.

**Return**

- ESP\_OK on success

`esp_err_t touch_pad_intr_disable ()`

To disable touch pad interrupt.

**Return**

- ESP\_OK on success

`esp_err_t touch_pad_set_filter_period (uint32_t new_period_ms)`

set touch pad filter calibration period, in ms. Need to call `touch_pad_filter_start` before all touch filter APIs

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_STATE driver state error
- ESP\_ERR\_INVALID\_ARG parameter error

**Parameters**

- `new_period_ms`: filter period, in ms

`esp_err_t touch_pad_get_filter_period (uint32_t *p_period_ms)`

get touch pad filter calibration period, in ms Need to call `touch_pad_filter_start` before all touch filter APIs

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_STATE driver state error
- ESP\_ERR\_INVALID\_ARG parameter error

**Parameters**

- `p_period_ms`: pointer to accept period

`esp_err_t touch_pad_filter_start (uint32_t filter_period_ms)`

start touch pad filter function This API will start a filter to process the noise in order to prevent false triggering when detecting slight change of capacitance. Need to call `touch_pad_filter_start` before all touch filter APIs

If filter is not initialized, this API will initialize the filter with given period. If filter is already initialized, this API will update the filter period.

**Note** This filter uses FreeRTOS timer, which is dispatched from a task with priority 1 by default on CPU 0. So if some application task with higher priority takes a lot of CPU0 time, then the quality of data obtained from this filter will be affected. You can adjust FreeRTOS timer task priority in `menuconfig`.

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_ARG parameter error
- ESP\_ERR\_NO\_MEM No memory for driver
- ESP\_ERR\_INVALID\_STATE driver state error

**Parameters**

- `filter_period_ms`: filter calibration period, in ms

`esp_err_t touch_pad_filter_stop ()`

stop touch pad filter function Need to call `touch_pad_filter_start` before all touch filter APIs

**Return**

- ESP\_OK Success
- ESP\_ERR\_INVALID\_STATE driver state error

`esp_err_t touch_pad_filter_delete ()`

delete touch pad filter driver and release the memory Need to call `touch_pad_filter_start` before all touch filter APIs

### Return

- ESP\_OK Success
- ESP\_ERR\_INVALID\_STATE driver state error

### Macros

#### **TOUCH\_PAD\_SLEEP\_CYCLE\_DEFAULT**

The timer frequency is RTC\_SLOW\_CLK (can be 150k or 32k depending on the options), max value is 0xffff

#### **TOUCH\_PAD\_MEASURE\_CYCLE\_DEFAULT**

The timer frequency is 8Mhz, the max value is 0xffff

#### **TOUCH\_FSM\_MODE\_DEFAULT**

The touch FSM may be started by the software or timer

#### **TOUCH\_TRIGGER\_MODE\_DEFAULT**

Interrupts can be triggered if sensor value gets below or above threshold

#### **TOUCH\_TRIGGER\_SOURCE\_DEFAULT**

The wakeup trigger source can be SET1 or both SET1 and SET2

#### **TOUCH\_PAD\_BIT\_MASK\_MAX**

### Type Definitions

```
typedef intr_handle_t touch_isr_handle_t
```

### Enumerations

```
enum touch_pad_t
```

*Values:*

```
TOUCH_PAD_NUM0 = 0
```

Touch pad channel 0 is GPIO4

```
TOUCH_PAD_NUM1
```

Touch pad channel 1 is GPIO0

```
TOUCH_PAD_NUM2
```

Touch pad channel 2 is GPIO2

```
TOUCH_PAD_NUM3
```

Touch pad channel 3 is GPIO15

```
TOUCH_PAD_NUM4
```

Touch pad channel 4 is GPIO13

```
TOUCH_PAD_NUM5
```

Touch pad channel 5 is GPIO12

```
TOUCH_PAD_NUM6
```

Touch pad channel 6 is GPIO14

```
TOUCH_PAD_NUM7
```

Touch pad channel 7 is GPIO27

**TOUCH\_PAD\_NUM8**

Touch pad channel 8 is GPIO33

**TOUCH\_PAD\_NUM9**

Touch pad channel 9 is GPIO32

**TOUCH\_PAD\_MAX****enum touch\_high\_volt\_t**

*Values:*

**TOUCH\_HVOLT\_KEEP = -1**

Touch sensor high reference voltage, no change

**TOUCH\_HVOLT\_2V4 = 0**

Touch sensor high reference voltage, 2.4V

**TOUCH\_HVOLT\_2V5**

Touch sensor high reference voltage, 2.5V

**TOUCH\_HVOLT\_2V6**

Touch sensor high reference voltage, 2.6V

**TOUCH\_HVOLT\_2V7**

Touch sensor high reference voltage, 2.7V

**TOUCH\_HVOLT\_MAX****enum touch\_low\_volt\_t**

*Values:*

**TOUCH\_LVOLT\_KEEP = -1**

Touch sensor low reference voltage, no change

**TOUCH\_LVOLT\_0V5 = 0**

Touch sensor low reference voltage, 0.5V

**TOUCH\_LVOLT\_0V6**

Touch sensor low reference voltage, 0.6V

**TOUCH\_LVOLT\_0V7**

Touch sensor low reference voltage, 0.7V

**TOUCH\_LVOLT\_0V8**

Touch sensor low reference voltage, 0.8V

**TOUCH\_LVOLT\_MAX****enum touch\_volt\_atten\_t**

*Values:*

**TOUCH\_HVOLT\_ATTEN\_KEEP = -1**

Touch sensor high reference voltage attenuation, no change

**TOUCH\_HVOLT\_ATTEN\_1V5 = 0**

Touch sensor high reference voltage attenuation, 1.5V attenuation

**TOUCH\_HVOLT\_ATTEN\_1V**

Touch sensor high reference voltage attenuation, 1.0V attenuation

**TOUCH\_HVOLT\_ATTEN\_0V5**

Touch sensor high reference voltage attenuation, 0.5V attenuation

**TOUCH\_HVOLT\_ATTEN\_0V**

Touch sensor high reference voltage attenuation, 0V attenuation

**TOUCH\_HVOLT\_ATTEN\_MAX**

**enum touch\_cnt\_slope\_t**

*Values:*

**TOUCH\_PAD\_SLOPE\_0 = 0**

Touch sensor charge / discharge speed, always zero

**TOUCH\_PAD\_SLOPE\_1 = 1**

Touch sensor charge / discharge speed, slowest

**TOUCH\_PAD\_SLOPE\_2 = 2**

Touch sensor charge / discharge speed

**TOUCH\_PAD\_SLOPE\_3 = 3**

Touch sensor charge / discharge speed

**TOUCH\_PAD\_SLOPE\_4 = 4**

Touch sensor charge / discharge speed

**TOUCH\_PAD\_SLOPE\_5 = 5**

Touch sensor charge / discharge speed

**TOUCH\_PAD\_SLOPE\_6 = 6**

Touch sensor charge / discharge speed

**TOUCH\_PAD\_SLOPE\_7 = 7**

Touch sensor charge / discharge speed, fast

**TOUCH\_PAD\_SLOPE\_MAX**

**enum touch\_trigger\_mode\_t**

*Values:*

**TOUCH\_TRIGGER\_BELOW = 0**

Touch interrupt will happen if counter value is less than threshold.

**TOUCH\_TRIGGER\_ABOVE = 1**

Touch interrupt will happen if counter value is larger than threshold.

**TOUCH\_TRIGGER\_MAX**

**enum touch\_trigger\_src\_t**

*Values:*

**TOUCH\_TRIGGER\_SOURCE\_BOTH = 0**

wakeup interrupt is generated if both SET1 and SET2 are “touched”

**TOUCH\_TRIGGER\_SOURCE\_SET1 = 1**

wakeup interrupt is generated if SET1 is “touched”

**TOUCH\_TRIGGER\_SOURCE\_MAX**

**enum touch\_tie\_opt\_t**

*Values:*

**TOUCH\_PAD\_TIE\_OPT\_LOW = 0**

Initial level of charging voltage, low level

**TOUCH\_PAD\_TIE\_OPT\_HIGH = 1**

Initial level of charging voltage, high level

**TOUCH\_PAD\_TIE\_OPT\_MAX**

```
enum touch_fsm_mode_t
```

*Values:*

```
TOUCH_FSM_MODE_TIMER = 0
```

To start touch FSM by timer

```
TOUCH_FSM_MODE_SW
```

To start touch FSM by software trigger

```
TOUCH_FSM_MODE_MAX
```

## GPIO Lookup Macros

Some useful macros can be used to specified the GPIO number of a touchpad channel, or vice versa. e.g.

1. TOUCH\_PAD\_NUM5\_GPIO\_NUM is the GPIO number of channel 5 (12);
2. TOUCH\_PAD\_GPIO4\_CHANNEL is the channel number of GPIO 4 (channel 0).

## Header File

- [soc/esp32/include/soc/touch\\_channel.h](#)

## Macros

```
TOUCH_PAD_GPIO4_CHANNEL
```

```
TOUCH_PAD_NUM0_GPIO_NUM
```

```
TOUCH_PAD_GPIO0_CHANNEL
```

```
TOUCH_PAD_NUM1_GPIO_NUM
```

```
TOUCH_PAD_GPIO2_CHANNEL
```

```
TOUCH_PAD_NUM2_GPIO_NUM
```

```
TOUCH_PAD_GPIO15_CHANNEL
```

```
TOUCH_PAD_NUM3_GPIO_NUM
```

```
TOUCH_PAD_GPIO13_CHANNEL
```

```
TOUCH_PAD_NUM4_GPIO_NUM
```

```
TOUCH_PAD_GPIO12_CHANNEL
```

```
TOUCH_PAD_NUM5_GPIO_NUM
```

```
TOUCH_PAD_GPIO14_CHANNEL
```

```
TOUCH_PAD_NUM6_GPIO_NUM
```

```
TOUCH_PAD_GPIO27_CHANNEL
```

```
TOUCH_PAD_NUM7_GPIO_NUM
```

```
TOUCH_PAD_GPIO33_CHANNEL
```

```
TOUCH_PAD_NUM8_GPIO_NUM
```

```
TOUCH_PAD_GPIO32_CHANNEL
```

TOUCH\_PAD\_NUM9\_GPIO\_NUM

## 2.4.16 UART

### Overview

An Universal Asynchronous Receiver/Transmitter (UART) is a component known to handle the timing requirements for a variety of widely-adapted protocols (RS232, RS485, RS422, ...). An UART provides a widely adopted and cheap method to realize full-duplex data exchange among different devices.

There are three UART controllers available on the ESP32 chip. They are compatible with UART-enabled devices from various manufacturers. All UART controllers integrated in the ESP32 feature an identical set of registers for ease of programming and flexibility. In this documentation, these controllers are referred to as UART0, UART1, and UART2.

### Functional Overview

The following overview describes functions and data types used to establish communication between ESP32 and some other UART device. The overview reflects a typical workflow when programming ESP32's UART driver and is broken down into the following sections:

1. *Setting Communication Parameters* - baud rate, data bits, stop bits, etc,
2. *Setting Communication Pins* - pins the other UART is connected to
3. *Driver Installation* - allocate ESP32's resources for the UART driver
4. *Running UART Communication* - send / receive the data
5. *Using Interrupts* - trigger interrupts on specific communication events
6. *Deleting Driver* - release ESP32's resources, if UART communication is not required anymore

The minimum to make the UART working is to complete the first four steps, the last two steps are optional.

The driver is identified by `uart_port_t`, that corresponds to one of the three UART controllers. Such identification is present in all the following function calls.

### Setting Communication Parameters

There are two ways to set the communications parameters for UART. One is to do it in one shot by calling `uart_param_config()` provided with configuration parameters in `uart_config_t` structure.

The alternate way is to configure specific parameters individually by calling dedicated functions:

- Baud rate - `uart_set_baudrate()`
- Number of transmitted bits - `uart_set_word_length()` selected out of `uart_word_length_t`
- Parity control - `uart_set_parity()` selected out of `uart_parity_t`
- Number of stop bits - `uart_set_stop_bits()` selected out of `uart_stop_bits_t`
- Hardware flow control mode - `uart_set_hw_flow_ctrl()` selected out of `uart_hw_flowcontrol_t`

All the above functions have a `_get_` equivalent to retrieve the current setting, e.g. `uart_get_baudrate()`.

## Setting Communication Pins

In next step, after configuring communication parameters, we are setting physical GPIO pin numbers the other UART will be connected to. This is done in a single step by calling function `uart_set_pin()` and providing it with GPIO numbers, that driver should use for the Tx, Rx, RTS and CTS signals.

Instead of GPIO pin number we can enter a macro `UART_PIN_NO_CHANGE` and the currently allocated pin will not be changed. The same macro should be entered if certain pin will not be used.

## Driver Installation

Once configuration of driver is complete, we can install it by calling `uart_driver_install()`. As result several resources required by the UART will be allocated. The type / size of resources are specified as function call parameters and concern:

- size of the send buffer
- size of the receive buffer
- the event queue handle and size
- flags to allocate an interrupt

If all above steps have been complete, we are ready to connect the other UART device and check the communication.

## Running UART Communication

The processes of serial communication are under control of UART's hardware FSM. The data to be sent should be put into Tx FIFO buffer, FSM will serialize them and sent out. A similar process, but in reverse order, is done to receive the data. Incoming serial stream is processed by FSM and moved to the Rx FIFO buffer. Therefore the task of API's communication functions is limited to writing and reading the data to / from the respective buffer. This is reflected in some function names, e.g.: `uart_write_bytes()` to transmit the data out, or `uart_read_bytes()` to read the incoming data.

## Transmitting

The basic API function to write the data to Tx FIFO buffer is `uart_tx_chars()`. If the buffer contains not sent characters, this function will write what fits into the empty space and exit reporting the number of bytes actually written.

There is a 'companion' function `uart_wait_tx_done()` that waits until all the data are transmitted out and the Tx FIFO is empty.

An easier to work with function is `uart_write_bytes()`. It sets up an intermediate ring buffer and exits after copying the data to this buffer. When there is an empty space in the FIFO, the data are moved from the ring buffer to the FIFO in the background by an ISR.

There is a similar function as above that adds a serial break signal after sending the data - `uart_write_bytes_with_break()`. The 'serial break signal' means holding TX line low for period longer than one data frame.

## Receiving

To retrieve the data received by UART and saved in Rx FIFO, use function `uart_read_bytes()`. You can check in advance what is the number of bytes available in Rx FIFO by calling `uart_get_buffered_data_len()`.

If the data in Rx FIFO is not required and should be discarded, call `uart_flush()`.

## Software Flow Control

When the hardware flow control is disabled, then use `uart_set_rts()` and `uart_set_dtr()` to manually set the levels of the RTS and DTR signals.

## Using Interrupts

There are nineteen interrupts reported on specific states of UART or on detected errors. The full list of available interrupts is described in [ESP32 Technical Reference Manual \(PDF\)](#). To enable specific interrupts call `uart_enable_intr_mask()`, to disable call `uart_disable_intr_mask()`. The mask of all interrupts is available as `UART_INTR_MASK`. Registration of an handler to service interrupts is done with `uart_isr_register()`, freeing the handler with `uart_isr_free()`. To clear the interrupt status bits once the handler is called use `uart_clear_intr_status()`.

The API provides a convenient way to handle specific interrupts discussed above by wrapping them into dedicated functions:

- **Event detection** - there are several events defined in `uart_event_type_t` that may be reported to user application using FreeRTOS queue functionality. You can enable this functionality when calling `uart_driver_install()` described in *Driver Installation*. Example how to use it is covered in [peripherals/uart\\_events](#).
- **FIFO space threshold or transmission timeout reached** - the interrupts on TX or Rx FIFO buffer being filled with specific number of characters or on a timeout of sending or receiving data. To use these interrupts, first configure respective threshold values of the buffer length and the timeout by entering them in `uart_intr_config_t` structure and calling `uart_intr_config()`. Then enable interrupts with functions `uart_enable_rx_intr()` and `uart_enable_tx_intr()`. To disable these interrupts there are corresponding functions `uart_disable_rx_intr()` or `uart_disable_tx_intr()`.
- **Pattern detection** - an interrupt triggered on detecting a 'pattern' of the same character being sent number of times. The functions that allow to configure, enable and disable this interrupt are `uart_enable_pattern_det_intr()` and `uart_disable_pattern_det_intr()`.

## Macros

The API provides several macros to define configuration parameters, e.g. `UART_FIFO_LEN` to define the length of the hardware FIFO buffers, `UART_BITRATE_MAX` that gives the maximum baud rate supported by UART, etc.

## Deleting Driver

If communication is established with `uart_driver_install()` for some specific period of time and then not required, the driver may be removed to free allocated resources by calling `uart_driver_delete()`.

## Application Examples

Configure UART settings and install UART driver to read/write using UART1 interface: [peripherals/uart\\_echo](#).

Demonstration of how to report various communication events and how to use pattern detection interrupts: [peripherals/uart\\_events](#).

Transmitting and receiving with the same UART in two separate FreeRTOS tasks: [peripherals/uart\\_async\\_rxtxtasks](#).

## API Reference

### Header File

- `driver/include/driver/uart.h`

### Functions

`esp_err_t uart_set_word_length(uart_port_t uart_num, uart_word_length_t data_bit)`  
Set UART data bits.

#### Return

- ESP\_OK Success
- ESP\_FAIL Parameter error

#### Parameters

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `data_bit`: UART data bits

`esp_err_t uart_get_word_length(uart_port_t uart_num, uart_word_length_t *data_bit)`  
Get UART data bits.

#### Return

- ESP\_FAIL Parameter error
- ESP\_OK Success, result will be put in (\*data\_bit)

#### Parameters

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `data_bit`: Pointer to accept value of UART data bits.

`esp_err_t uart_set_stop_bits(uart_port_t uart_num, uart_stop_bits_t stop_bits)`  
Set UART stop bits.

#### Return

- ESP\_OK Success
- ESP\_FAIL Fail

#### Parameters

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2

- `stop_bits`: UART stop bits

`esp_err_t uart_get_stop_bits (uart_port_t uart_num, uart_stop_bits_t *stop_bits)`  
Get UART stop bits.

**Return**

- `ESP_FAIL` Parameter error
- `ESP_OK` Success, result will be put in (`*stop_bit`)

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `stop_bits`: Pointer to accept value of UART stop bits.

`esp_err_t uart_set_parity (uart_port_t uart_num, uart_parity_t parity_mode)`  
Set UART parity mode.

**Return**

- `ESP_FAIL` Parameter error
- `ESP_OK` Success

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `parity_mode`: the enum of uart parity configuration

`esp_err_t uart_get_parity (uart_port_t uart_num, uart_parity_t *parity_mode)`  
Get UART parity mode.

**Return**

- `ESP_FAIL` Parameter error
- `ESP_OK` Success, result will be put in (`*parity_mode`)

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `parity_mode`: Pointer to accept value of UART parity mode.

`esp_err_t uart_set_baudrate (uart_port_t uart_num, uint32_t baudrate)`  
Set UART baud rate.

**Return**

- `ESP_FAIL` Parameter error
- `ESP_OK` Success

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `baudrate`: UART baud rate.

`esp_err_t uart_get_baudrate (uart_port_t uart_num, uint32_t *baudrate)`  
Get UART baud rate.

**Return**

- ESP\_FAIL Parameter error
- ESP\_OK Success, result will be put in (\*baudrate)

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `baudrate`: Pointer to accept value of UART baud rate

`esp_err_t uart_set_line_inverse` (*uart\_port\_t* `uart_num`, `uint32_t` `inverse_mask`)  
Set UART line inverse mode.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `inverse_mask`: Choose the wires that need to be inverted. `inverse_mask` should be chosen from `UART_INVERSE_RXD` / `UART_INVERSE_TXD` / `UART_INVERSE_RTS` / `UART_INVERSE_CTS`, combined with OR operation.

`esp_err_t uart_set_hw_flow_ctrl` (*uart\_port\_t* `uart_num`, *uart\_hw\_flowcontrol\_t* `flow_ctrl`, `uint8_t` `rx_thresh`)

Set hardware flow control.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `flow_ctrl`: Hardware flow control mode
- `rx_thresh`: Threshold of Hardware RX flow control (0 ~ `UART_FIFO_LEN`). Only when `UART_HW_FLOWCTRL_RTS` is set, will the `rx_thresh` value be set.

`esp_err_t uart_set_sw_flow_ctrl` (*uart\_port\_t* `uart_num`, `bool` `enable`, `uint8_t` `rx_thresh_xon`, `uint8_t` `rx_thresh_xoff`)

Set software flow control.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `enable`: switch on or off
- `rx_thresh_xon`: low water mark

- `rx_thresh_xoff`: high water mark

`esp_err_t uart_get_hw_flow_ctrl (uart_port_t uart_num, uart_hw_flowcontrol_t *flow_ctrl)`  
Get hardware flow control mode.

**Return**

- `ESP_FAIL` Parameter error
- `ESP_OK` Success, result will be put in (`*flow_ctrl`)

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `flow_ctrl`: Option for different flow control mode.

`esp_err_t uart_clear_intr_status (uart_port_t uart_num, uint32_t clr_mask)`  
Clear UART interrupt status.

**Return**

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `clr_mask`: Bit mask of the interrupt status to be cleared. The bit mask should be composed from the fields of register `UART_INT_CLR_REG`.

`esp_err_t uart_enable_intr_mask (uart_port_t uart_num, uint32_t enable_mask)`  
Set UART interrupt enable.

**Return**

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `enable_mask`: Bit mask of the enable bits. The bit mask should be composed from the fields of register `UART_INT_ENA_REG`.

`esp_err_t uart_disable_intr_mask (uart_port_t uart_num, uint32_t disable_mask)`  
Clear UART interrupt enable bits.

**Return**

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `disable_mask`: Bit mask of the disable bits. The bit mask should be composed from the fields of register `UART_INT_ENA_REG`.

`esp_err_t uart_enable_rx_intr (uart_port_t uart_num)`  
 Enable UART RX interrupt (RX\_FULL & RX\_TIMEOUT INTERRUPT)

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2

`esp_err_t uart_disable_rx_intr (uart_port_t uart_num)`  
 Disable UART RX interrupt (RX\_FULL & RX\_TIMEOUT INTERRUPT)

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2

`esp_err_t uart_disable_tx_intr (uart_port_t uart_num)`  
 Disable UART TX interrupt (TX\_FULL & TX\_TIMEOUT INTERRUPT)

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2

`esp_err_t uart_enable_tx_intr (uart_port_t uart_num, int enable, int thresh)`  
 Enable UART TX interrupt (TX\_FULL & TX\_TIMEOUT INTERRUPT)

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `enable`: 1: enable; 0: disable
- `thresh`: Threshold of TX interrupt, 0 ~ UART\_FIFO\_LEN

`esp_err_t uart_isr_register (uart_port_t uart_num, void (*fn) void *, void *arg, int intr_alloc_flags, uart_isr_handle_t *handle)` Register UART interrupt handler (ISR).

**Note** UART ISR handler will be attached to the same CPU core that this function is running on.

**Return**

- ESP\_OK Success

- ESP\_FAIL Parameter error

#### Parameters

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `fn`: Interrupt handler function.
- `arg`: parameter for handler function
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) ESP\_INTR\_FLAG\_\* values. See `esp_intr_alloc.h` for more info.
- `handle`: Pointer to return handle. If non-NULL, a handle for the interrupt will be returned here.

`esp_err_t uart_isr_free` (*uart\_port\_t* `uart_num`)

Free UART interrupt handler registered by `uart_isr_register`. Must be called on the same core as `uart_isr_register` was called.

#### Return

- ESP\_OK Success
- ESP\_FAIL Parameter error

#### Parameters

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2

`esp_err_t uart_set_pin` (*uart\_port\_t* `uart_num`, `tx_io_num`, `rx_io_num`, `rts_io_num`, `cts_io_num`)

Set UART pin number.

**Note** Internal signal can be output to multiple GPIO pads. Only one GPIO pad can connect with input signal.

**Note** Instead of GPIO number a macro 'UART\_PIN\_NO\_CHANGE' may be provided to keep the currently allocated pin.

#### Return

- ESP\_OK Success
- ESP\_FAIL Parameter error

#### Parameters

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `tx_io_num`: UART TX pin GPIO number.
- `rx_io_num`: UART RX pin GPIO number.
- `rts_io_num`: UART RTS pin GPIO number.
- `cts_io_num`: UART CTS pin GPIO number.

`esp_err_t uart_set_rts` (*uart\_port\_t* `uart_num`, `int level`)

Manually set the UART RTS pin level.

**Note** UART must be configured with hardware flow control disabled.

#### Return

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `level`: 1: RTS output low (active); 0: RTS output high (block)

`esp_err_t uart_set_dtr(uart_port_t uart_num, int level)`

Manually set the UART DTR pin level.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `level`: 1: DTR output low; 0: DTR output high

`esp_err_t uart_set_tx_idle_num(uart_port_t uart_num, uint16_t idle_num)`

Set UART idle interval after tx FIFO is empty.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `idle_num`: idle interval after tx FIFO is empty(unit: the time it takes to send one bit under current baudrate)

`esp_err_t uart_param_config(uart_port_t uart_num, const uart_config_t *uart_config)`

Set UART configuration parameters.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- `uart_config`: UART parameter settings

`esp_err_t uart_intr_config(uart_port_t uart_num, const uart_intr_config_t *intr_conf)`

Configure UART interrupts.

**Return**

- ESP\_OK Success
- ESP\_FAIL Parameter error

**Parameters**

- `uart_num`: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2

- `intr_conf`: UART interrupt settings

`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.

UART ISR handler will be attached to the same CPU core that this function is running on.

**Note** `Rx_buffer_size` should be greater than `UART_FIFO_LEN`. `Tx_buffer_size` should be either zero or greater than `UART_FIFO_LEN`.

#### Return

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

#### Parameters

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `rx_buffer_size`: UART RX ring buffer size.
- `tx_buffer_size`: UART TX ring buffer size. If set to zero, driver will not use TX buffer, TX function will block task until all data have been sent out.
- `queue_size`: UART event queue size/depth.
- `uart_queue`: UART event queue handle (out param). On success, a new queue handle is written here to provide access to UART events. If set to `NULL`, driver will not use an event queue.
- `intr_alloc_flags`: Flags used to allocate the interrupt. One or multiple (ORred) `ESP_INTR_FLAG_*` values. See `esp_intr_alloc.h` for more info. Do not set `ESP_INTR_FLAG_IRAM` here (the driver's ISR handler is not located in IRAM)

`esp_err_t uart_driver_delete` (*uart\_port\_t* `uart_num`)

Uninstall UART driver.

#### Return

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

#### Parameters

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`

`esp_err_t uart_wait_tx_done` (*uart\_port\_t* `uart_num`, `TickType_t` `ticks_to_wait`)

Wait until UART TX FIFO is empty.

#### Return

- `ESP_OK` Success
- `ESP_FAIL` Parameter error
- `ESP_ERR_TIMEOUT` Timeout

#### Parameters

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `ticks_to_wait`: Timeout, count in RTOS ticks

int **uart\_tx\_chars** (*uart\_port\_t* *uart\_num*, **const** char \**buffer*, uint32\_t *len*)

Send data to the UART port from a given buffer and length.

This function will not wait for enough space in TX FIFO. It will just fill the available TX FIFO and return when the FIFO is full.

**Note** This function should only be used when UART TX buffer is not enabled.

#### Return

- (-1) Parameter error
- OTHERS ( $\geq 0$ ) The number of bytes pushed to the TX FIFO

#### Parameters

- *uart\_num*: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- *buffer*: data buffer address
- *len*: data length to send

int **uart\_write\_bytes** (*uart\_port\_t* *uart\_num*, **const** char \**src*, size\_t *size*)

Send data to the UART port from a given buffer and length.

If the UART driver's parameter 'tx\_buffer\_size' is set to zero: This function will not return until all the data have been sent out, or at least pushed into TX FIFO.

Otherwise, if the 'tx\_buffer\_size'  $> 0$ , this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually.

#### Return

- (-1) Parameter error
- OTHERS ( $\geq 0$ ) The number of bytes pushed to the TX FIFO

#### Parameters

- *uart\_num*: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- *src*: data buffer address
- *size*: data length to send

int **uart\_write\_bytes\_with\_break** (*uart\_port\_t* *uart\_num*, **const** char \**src*, size\_t *size*, int *brk\_len*)

Send data to the UART port from a given buffer and length.

If the UART driver's parameter 'tx\_buffer\_size' is set to zero: This function will not return until all the data and the break signal have been sent out. After all data is sent out, send a break signal.

Otherwise, if the 'tx\_buffer\_size'  $> 0$ , this function will return after copying all the data to tx ring buffer, UART ISR will then move data from the ring buffer to TX FIFO gradually. After all data sent out, send a break signal.

#### Return

- (-1) Parameter error
- OTHERS ( $\geq 0$ ) The number of bytes pushed to the TX FIFO

#### Parameters

- *uart\_num*: UART\_NUM\_0, UART\_NUM\_1 or UART\_NUM\_2
- *src*: data buffer address
- *size*: data length to send

- `brk_len`: break signal length (unit: the time it takes to send a complete byte including start, stop and parity bits at `current_baudrate`)

`int uart_read_bytes` (*uart\_port\_t* `uart_num`, `uint8_t *buf`, `uint32_t length`, `TickType_t ticks_to_wait`)  
UART read bytes from UART buffer.

**Return**

- (-1) Error
- OTHERS (>=0) The number of bytes read from UART FIFO

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`
- `buf`: pointer to the buffer.
- `length`: data length
- `ticks_to_wait`: sTimeout, count in RTOS ticks

`esp_err_t uart_flush` (*uart\_port\_t* `uart_num`)  
Alias of `uart_flush_input`. UART ring buffer flush. This will discard all data in the UART RX buffer.

**Note** Instead of waiting the data sent out, this function will clear UART rx buffer. In order to send all the data in tx FIFO, we can use `uart_wait_tx_done` function.

**Return**

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`

`esp_err_t uart_flush_input` (*uart\_port\_t* `uart_num`)  
Clear input buffer, discard all the data is in the ring-buffer.

**Note** In order to send all the data in tx FIFO, we can use `uart_wait_tx_done` function.

**Return**

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

**Parameters**

- `uart_num`: `UART_NUM_0`, `UART_NUM_1` or `UART_NUM_2`

`esp_err_t uart_get_buffered_data_len` (*uart\_port\_t* `uart_num`, `size_t *size`)  
UART get RX ring buffer cached data length.

**Return**

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

**Parameters**

- `uart_num`: UART port number.
- `size`: Pointer of `size_t` to accept cached data length

`esp_err_t uart_disable_pattern_det_intr (uart_port_t uart_num)`

UART disable pattern detect function. Designed for applications like ‘AT commands’. When the hardware detects a series of one same character, the interrupt will be triggered.

#### Return

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

#### Parameters

- `uart_num`: UART port number.

`esp_err_t uart_enable_pattern_det_intr (uart_port_t uart_num, char pattern_chr, uint8_t chr_num, int chr_tout, int post_idle, int pre_idle)`

UART enable pattern detect function. Designed for applications like ‘AT commands’. When the hardware detect a series of one same character, the interrupt will be triggered.

#### Return

- `ESP_OK` Success
- `ESP_FAIL` Parameter error

#### Parameters

- `uart_num`: UART port number.
- `pattern_chr`: character of the pattern
- `chr_num`: number of the character, 8bit value.
- `chr_tout`: timeout of the interval between each pattern characters, 24bit value, unit is APB (80Mhz) clock cycle. When the duration is less than this value, it will not take this data as `at_cmd` char
- `post_idle`: idle time after the last pattern character, 24bit value, unit is APB (80Mhz) clock cycle. 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, 24bit value, unit is APB (80Mhz) clock cycle. When the duration is less than this value, it will not take this data as the first `at_cmd` char

`int uart_pattern_pop_pos (uart_port_t uart_num)`

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, this function will dequeue the first pattern position and move the pointer to next pattern position.

The following APIs will modify the pattern position info: `uart_flush_input`, `uart_read_bytes`, `uart_driver_delete`, `uart_pop_pattern_pos` It is the application’s responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

**Note** If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

#### Return

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

#### Parameters

- `uart_num`: UART port number

`int uart_pattern_get_pos (uart_port_t uart_num)`

Return the nearest detected pattern position in buffer. The positions of the detected pattern are saved in a queue, This function do nothing to the queue.

The following APIs will modify the pattern position info: `uart_flush_input`, `uart_read_bytes`, `uart_driver_delete`, `uart_pop_pattern_pos` It is the application's responsibility to ensure atomic access to the pattern queue and the rx data buffer when using pattern detect feature.

**Note** If the RX buffer is full and flow control is not enabled, the detected pattern may not be found in the rx buffer due to overflow.

#### Return

- (-1) No pattern found for current index or parameter error
- others the pattern position in rx buffer.

#### Parameters

- `uart_num`: UART port number

`esp_err_t uart_pattern_queue_reset (uart_port_t uart_num, int queue_length)`

Allocate a new memory with the given length to save record the detected pattern position in rx buffer.

#### Return

- `ESP_ERR_NO_MEM` No enough memory
- `ESP_ERR_INVALID_STATE` Driver not installed
- `ESP_FAIL` Parameter error
- `ESP_OK` Success

#### Parameters

- `uart_num`: UART port number
- `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.

## 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

bool **use\_ref\_tick**

Set to true if UART should be clocked from REF\_TICK

**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

### Macros

**UART\_FIFO\_LEN**

Length of the hardware FIFO buffers

**UART\_INTR\_MASK**

Mask of all UART interrupts

**UART\_LINE\_INV\_MASK**

TBD

**UART\_BITRATE\_MAX**

Max bit rate supported by UART

**UART\_PIN\_NO\_CHANGE**

Constant for `uart_set_pin` function which indicates that UART pin should not be changed

**UART\_INVERSE\_DISABLE**

Disable UART signal inverse

**UART\_INVERSE\_RXD**

UART RXD input inverse

**UART\_INVERSE\_CTS**

UART CTS input inverse

**UART\_INVERSE\_TXD**

UART TXD output inverse

**UART\_INVERSE\_RTS**

UART RTS output inverse

## Type Definitions

```
typedef intr_handle_t uart_isr_handle_t
```

## Enumerations

**enum uart\_word\_length\_t**

UART word length constants.

*Values:*

**UART\_DATA\_5\_BITS** = 0x0

word length: 5bits

**UART\_DATA\_6\_BITS** = 0x1

word length: 6bits

**UART\_DATA\_7\_BITS** = 0x2

word length: 7bits

**UART\_DATA\_8\_BITS** = 0x3

word length: 8bits

**UART\_DATA\_BITS\_MAX** = 0x4

**enum uart\_stop\_bits\_t**

UART stop bits number.

*Values:*

**UART\_STOP\_BITS\_1** = 0x1

stop bit: 1bit

**UART\_STOP\_BITS\_1\_5** = 0x2

stop bit: 1.5bits

**UART\_STOP\_BITS\_2** = 0x3

stop bit: 2bits

**UART\_STOP\_BITS\_MAX** = 0x4

**enum uart\_port\_t**

UART peripheral number.

*Values:*

**UART\_NUM\_0** = 0x0  
UART base address 0x3ff40000

**UART\_NUM\_1** = 0x1  
UART base address 0x3ff50000

**UART\_NUM\_2** = 0x2  
UART base address 0x3ff6e000

**UART\_NUM\_MAX**

**enum uart\_parity\_t**

UART parity constants.

*Values:*

**UART\_PARITY\_DISABLE** = 0x0  
Disable UART parity

**UART\_PARITY\_EVEN** = 0x2  
Enable UART even parity

**UART\_PARITY\_ODD** = 0x3  
Enable UART odd parity

**enum uart\_hw\_flowcontrol\_t**

UART hardware flow control modes.

*Values:*

**UART\_HW\_FLOWCTRL\_DISABLE** = 0x0  
disable hardware flow control

**UART\_HW\_FLOWCTRL\_RTS** = 0x1  
enable RX hardware flow control (rts)

**UART\_HW\_FLOWCTRL\_CTS** = 0x2  
enable TX hardware flow control (cts)

**UART\_HW\_FLOWCTRL\_CTS\_RTS** = 0x3  
enable hardware flow control

**UART\_HW\_FLOWCTRL\_MAX** = 0x4

**enum uart\_event\_type\_t**

UART event types used in the ring buffer.

*Values:*

**UART\_DATA**  
UART data event

**UART\_BREAK**  
UART break event

**UART\_BUFFER\_FULL**  
UART RX buffer full event

**UART\_FIFO\_OVF**  
UART FIFO overflow event

**UART\_FRAME\_ERR**  
UART RX frame error event

**UART\_PARITY\_ERR**

UART RX parity event

**UART\_DATA\_BREAK**

UART TX data and break event

**UART\_PATTERN\_DET**

UART pattern detected

**UART\_EVENT\_MAX**

UART event max index

## GPIO Lookup Macros

You can use macros to specify the **direct** GPIO (UART module connected to pads through direct IO mux without the GPIO mux) number of a UART channel, or vice versa. The pin name can be omitted if the channel of a GPIO number is specified, e.g.:

1. `UART_NUM_2_TXD_DIRECT_GPIO_NUM` is the GPIO number of UART channel 2 TXD pin (17);
2. `UART_GPIO19_DIRECT_CHANNEL` is the UART channel number of GPIO 19 (channel 0);
3. `UART_CTS_GPIO19_DIRECT_CHANNEL` is the UART channel number of GPIO 19, and GPIO 19 must be a CTS pin (channel 0).

## Header File

- `soc/esp32/include/soc/uart_channel.h`

## Macros

`UART_GPIO1_DIRECT_CHANNEL`

`UART_NUM_0_TXD_DIRECT_GPIO_NUM`

`UART_GPIO3_DIRECT_CHANNEL`

`UART_NUM_0_RXD_DIRECT_GPIO_NUM`

`UART_GPIO19_DIRECT_CHANNEL`

`UART_NUM_0_CTS_DIRECT_GPIO_NUM`

`UART_GPIO22_DIRECT_CHANNEL`

`UART_NUM_0_RTS_DIRECT_GPIO_NUM`

`UART_TXD_GPIO1_DIRECT_CHANNEL`

`UART_RXD_GPIO3_DIRECT_CHANNEL`

`UART_CTS_GPIO19_DIRECT_CHANNEL`

`UART_RTS_GPIO22_DIRECT_CHANNEL`

`UART_GPIO10_DIRECT_CHANNEL`

`UART_NUM_1_TXD_DIRECT_GPIO_NUM`

`UART_GPIO9_DIRECT_CHANNEL`

```
UART_NUM_1_RXD_DIRECT_GPIO_NUM
UART_GPIO6_DIRECT_CHANNEL
UART_NUM_1_CTS_DIRECT_GPIO_NUM
UART_GPIO11_DIRECT_CHANNEL
UART_NUM_1_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO10_DIRECT_CHANNEL
UART_RXD_GPIO9_DIRECT_CHANNEL
UART_CTS_GPIO6_DIRECT_CHANNEL
UART_RTS_GPIO11_DIRECT_CHANNEL
UART_GPIO17_DIRECT_CHANNEL
UART_NUM_2_TXD_DIRECT_GPIO_NUM
UART_GPIO16_DIRECT_CHANNEL
UART_NUM_2_RXD_DIRECT_GPIO_NUM
UART_GPIO8_DIRECT_CHANNEL
UART_NUM_2_CTS_DIRECT_GPIO_NUM
UART_GPIO7_DIRECT_CHANNEL
UART_NUM_2_RTS_DIRECT_GPIO_NUM
UART_TXD_GPIO17_DIRECT_CHANNEL
UART_RXD_GPIO16_DIRECT_CHANNEL
UART_CTS_GPIO8_DIRECT_CHANNEL
UART_RTS_GPIO7_DIRECT_CHANNEL
```

Example code for this API section is provided in [peripherals](#) directory of ESP-IDF examples.

## 2.5 Protocols API

### 2.5.1 mDNS Service

#### Overview

mDNS is a multicast UDP service that is used to provide local network service and host discovery.

mDNS is installed by default on most operating systems or is available as separate package. On Mac OS it is installed by default and is called `Bonjour`. Apple releases an installer for Windows that can be found on [Apple's support page](#). On Linux, mDNS is provided by `avahi` and is usually installed by default.

#### mDNS Properties

- `hostname`: the hostname that the device will respond to. If not set, the `hostname` will be read from the interface. Example: `my-esp32` will resolve to `my-esp32.local`

- `default_instance`: friendly name for your device, like Jhon's ESP32 Thing. If not set, hostname will be used.

Example method to start mDNS for the STA interface and set hostname and `default_instance`:

```
void start_mdns_service()
{
    //initialize mDNS service
    esp_err_t err = mdns_init();
    if (err) {
        printf("MDNS Init failed: %d\n", err);
        return;
    }

    //set hostname
    mdns_hostname_set("my-esp32");
    //set default instance
    mdns_instance_name_set("Jhon's ESP32 Thing");
}
```

## mDNS Services

mDNS can advertise information about network services that your device offers. Each service is defined by a few properties.

- `instance_name`: friendly name for your service, like Jhon's ESP32 Web Server. If not defined, `default_instance` will be used.
- `service_type`: (required) service type, prepended with underscore. Some common types can be found [here](#).
- `proto`: (required) protocol that the service runs on, prepended with underscore. Example: `_tcp` or `_udp`
- `port`: (required) network port that the service runs on
- `txt`: {var, val} array of strings, used to define properties for your service

Example method to add a few services and different properties:

```
void add_mdns_services()
{
    //add our services
    mdns_service_add(NULL, "_http", "_tcp", 80, NULL, 0);
    mdns_service_add(NULL, "_arduino", "_tcp", 3232, NULL, 0);
    mdns_service_add(NULL, "_myservice", "_udp", 1234, NULL, 0);

    //NOTE: services must be added before their properties can be set
    //use custom instance for the web server
    mdns_service_instance_name_set("_http", "_tcp", "Jhon's ESP32 Web Server");

    mdns_txt_item_t serviceTxtData[3] = {
        {"board", "esp32"},
        {"u", "user"},
        {"p", "password"}
    };
    //set txt data for service (will free and replace current data)
    mdns_service_txt_set("_http", "_tcp", serviceTxtData, 3);

    //change service port
```

(continues on next page)

(continued from previous page)

```
mdns_service_port_set("_myservice", "udp", 4321);
}
```

## mDNS Query

mDNS provides methods for browsing for services and resolving host's IP/IPv6 addresses.

Results for services are returned as a linked list of `mdns_result_t` objects.

Example method to resolve host IPs:

```
void resolve_mdns_host(const char * host_name)
{
    printf("Query A: %s.local", host_name);

    struct ip4_addr addr;
    addr.addr = 0;

    esp_err_t err = mdns_query_a(host_name, 2000, &addr);
    if(err){
        if(err == ESP_ERR_NOT_FOUND){
            printf("Host was not found!");
            return;
        }
        printf("Query Failed");
        return;
    }

    printf(IPSTR, IP2STR(&addr));
}
```

Example method to resolve local services:

```
static const char * if_str[] = {"STA", "AP", "ETH", "MAX"};
static const char * ip_protocol_str[] = {"V4", "V6", "MAX"};

void mdns_print_results(mdns_result_t * results){
    mdns_result_t * r = results;
    mdns_ip_addr_t * a = NULL;
    int i = 1, t;
    while(r){
        printf("%d: Interface: %s, Type: %s\n", i++, if_str[r->tcpip_if], ip_protocol_
↪str[r->ip_protocol]);
        if(r->instance_name){
            printf(" PTR : %s\n", r->instance_name);
        }
        if(r->hostname){
            printf(" SRV : %s.local:%u\n", r->hostname, r->port);
        }
        if(r->txt_count){
            printf(" TXT : [%u] ", r->txt_count);
            for(t=0; t<r->txt_count; t++){
                printf("%s=%s; ", r->txt[t].key, r->txt[t].value);
            }
            printf("\n");
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    a = r->addr;
    while(a){
        if(a->addr.type == MDNS_IP_PROTOCOL_V6){
            printf(" AAAA: " IPV6STR "\n", IPV62STR(a->addr.u_addr.ip6));
        } else {
            printf(" A   : " IPSTR "\n", IP2STR(&(a->addr.u_addr.ip4)));
        }
        a = a->next;
    }
    r = r->next;
}

void find_mdns_service(const char * service_name, const char * proto)
{
    ESP_LOGI(TAG, "Query PTR: %s.%s.local", service_name, proto);

    mdns_result_t * results = NULL;
    esp_err_t err = mdns_query_ptr(service_name, proto, 3000, 20, &results);
    if(err){
        ESP_LOGE(TAG, "Query Failed");
        return;
    }
    if(!results){
        ESP_LOGW(TAG, "No results found!");
        return;
    }

    mdns_print_results(results);
    mdns_query_results_free(results);
}

```

Example of using the methods above:

```

void my_app_some_method(){
    //search for esp32-mdns.local
    resolve_mdns_host("esp32-mdns");

    //search for HTTP servers
    find_mdns_service("_http", "_tcp");
    //or file servers
    find_mdns_service("_smb", "_tcp"); //windows sharing
    find_mdns_service("_afpovertcp", "_tcp"); //apple sharing
    find_mdns_service("_nfs", "_tcp"); //NFS server
    find_mdns_service("_ftp", "_tcp"); //FTP server
    //or networked printer
    find_mdns_service("_printer", "_tcp");
    find_mdns_service("_ipp", "_tcp");
}

```

## Application Example

mDNS server/scanner example: [protocols/mdns](#).

## API Reference

### Header File

- `mdns/include/mdns.h`

### Functions

`esp_err_t mdns_init ()`  
Initialize mDNS on given interface.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` when bad `tcpip_if` is given
- `ESP_ERR_INVALID_STATE` when the network returned error
- `ESP_ERR_NO_MEM` on memory error
- `ESP_ERR_WIFI_NOT_INIT` when WiFi is not initialized by `eps_wifi_init`

`void mdns_free ()`  
Stop and free mDNS server.

`esp_err_t mdns_hostname_set (const char *hostname)`  
Set the hostname for mDNS server required if you want to advertise services.

#### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_ERR_NO_MEM` memory error

#### Parameters

- `hostname`: Hostname to set

`esp_err_t mdns_instance_name_set (const char *instance_name)`  
Set the default instance name for mDNS server.

#### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_ARG` Parameter error
- `ESP_ERR_NO_MEM` memory error

#### Parameters

- `instance_name`: Instance name to set

`esp_err_t mdns_service_add (const char *instance_name, const char *service_type, const char *proto, uint16_t port, mdns_txt_item_t txt[], size_t num_items)`  
Add service to mDNS server.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NO\_MEM memory error

#### Parameters

- `instance_name`: instance name to set. If NULL, global instance name or hostname will be used
- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)
- `port`: service port
- `num_items`: number of items in TXT data
- `txt`: string array of TXT data (eg. `{ {"var","val"}, {"other","2"} }`)

`esp_err_t mdns_service_remove (const char *service_type, const char *proto)`  
Remove service from mDNS server.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NOT\_FOUND Service not found
- ESP\_FAIL unknown error

#### Parameters

- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)

`esp_err_t mdns_service_instance_name_set (const char *service_type, const char *proto, const char *instance_name)`

Set instance name for service.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NOT\_FOUND Service not found
- ESP\_ERR\_NO\_MEM memory error

#### Parameters

- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)
- `instance_name`: instance name to set

`esp_err_t mdns_service_port_set (const char *service_type, const char *proto, uint16_t port)`  
Set service port.

#### Return

- ESP\_OK success

- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NOT\_FOUND Service not found

#### Parameters

- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)
- `port`: service port

`esp_err_t mdns_service_txt_set` (`const char *service_type`, `const char *proto`, `mdns_txt_item_t txt[]`, `uint8_t num_items`)

Replace all TXT items for service.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NOT\_FOUND Service not found
- ESP\_ERR\_NO\_MEM memory error

#### Parameters

- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)
- `num_items`: number of items in TXT data
- `txt`: array of TXT data (eg. `{{"var","val"}, {"other","2"}}`)

`esp_err_t mdns_service_txt_item_set` (`const char *service_type`, `const char *proto`, `const char *key`, `const char *value`)

Set/Add TXT item for service TXT record.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NOT\_FOUND Service not found
- ESP\_ERR\_NO\_MEM memory error

#### Parameters

- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)
- `key`: the key that you want to add/update
- `value`: the new value of the key

`esp_err_t mdns_service_txt_item_remove` (`const char *service_type`, `const char *proto`, `const char *key`)

Remove TXT item for service TXT record.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error
- ESP\_ERR\_NOT\_FOUND Service not found
- ESP\_ERR\_NO\_MEM memory error

#### Parameters

- `service_type`: service type (`_http`, `_ftp`, etc)
- `proto`: service protocol (`_tcp`, `_udp`)
- `key`: the key that you want to remove

`esp_err_t mdns_service_remove_all ()`

Remove and free all services from mDNS server.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_ARG Parameter error

`esp_err_t mdns_query (const char *name, const char *service_type, const char *proto, uint16_t type, uint32_t timeout, size_t max_results, mdns_result_t **results)`

Query mDNS for host or service All following query methods are derived from this one.

#### Return

- ESP\_OK success
- ESP\_ERR\_INVALID\_STATE mDNS is not running
- ESP\_ERR\_NO\_MEM memory error
- ESP\_ERR\_INVALID\_ARG timeout was not given

#### Parameters

- `name`: service instance or host name (NULL for PTR queries)
- `service_type`: service type (`_http`, `_arduino`, `_ftp` etc.) (NULL for host queries)
- `proto`: service protocol (`_tcp`, `_udp`, etc.) (NULL for host queries)
- `type`: type of query (`MDNS_TYPE_*`)
- `timeout`: time in milliseconds to wait for answers.
- `max_results`: maximum results to be collected
- `results`: pointer to the results of the query results must be freed using `mdns_query_results_free` below

`void mdns_query_results_free (mdns_result_t *results)`

Free query results.

#### Parameters

- `results`: linked list of results to be freed

`esp_err_t mdns_query_ptr` (`const char *service_type`, `const char *proto`, `uint32_t timeout`, `size_t max_results`, `mdns_result_t **results`)  
Query mDNS for service.

**Return**

- `ESP_OK` success
- `ESP_ERR_INVALID_STATE` mDNS is not running
- `ESP_ERR_NO_MEM` memory error
- `ESP_ERR_INVALID_ARG` parameter error

**Parameters**

- `service_type`: service type (`_http`, `_arduino`, `_ftp` etc.)
- `proto`: service protocol (`_tcp`, `_udp`, etc.)
- `timeout`: time in milliseconds to wait for answer.
- `max_results`: maximum results to be collected
- `results`: pointer to the results of the query

`esp_err_t mdns_query_srv` (`const char *instance_name`, `const char *service_type`, `const char *proto`, `uint32_t timeout`, `mdns_result_t **result`)  
Query mDNS for SRV record.

**Return**

- `ESP_OK` success
- `ESP_ERR_INVALID_STATE` mDNS is not running
- `ESP_ERR_NO_MEM` memory error
- `ESP_ERR_INVALID_ARG` parameter error

**Parameters**

- `instance_name`: service instance name
- `service_type`: service type (`_http`, `_arduino`, `_ftp` etc.)
- `proto`: service protocol (`_tcp`, `_udp`, etc.)
- `timeout`: time in milliseconds to wait for answer.
- `result`: pointer to the result of the query

`esp_err_t mdns_query_txt` (`const char *instance_name`, `const char *service_type`, `const char *proto`, `uint32_t timeout`, `mdns_result_t **result`)  
Query mDNS for TXT record.

**Return**

- `ESP_OK` success
- `ESP_ERR_INVALID_STATE` mDNS is not running
- `ESP_ERR_NO_MEM` memory error
- `ESP_ERR_INVALID_ARG` parameter error

**Parameters**

- `instance_name`: service instance name
- `service_type`: service type (`_http`, `_arduino`, `_ftp` etc.)
- `proto`: service protocol (`_tcp`, `_udp`, etc.)
- `timeout`: time in milliseconds to wait for answer.
- `result`: pointer to the result of the query

`esp_err_t mdns_query_a (const char *host_name, uint32_t timeout, ip4_addr_t *addr)`  
Query mDNS for A record.

#### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_STATE` mDNS is not running
- `ESP_ERR_NO_MEM` memory error
- `ESP_ERR_INVALID_ARG` parameter error

#### Parameters

- `host_name`: host name to look for
- `timeout`: time in milliseconds to wait for answer.
- `addr`: pointer to the resulting IP4 address

`esp_err_t mdns_query_aaaa (const char *host_name, uint32_t timeout, ip6_addr_t *addr)`  
Query mDNS for AAAA record.

#### Return

- `ESP_OK` success
- `ESP_ERR_INVALID_STATE` mDNS is not running
- `ESP_ERR_NO_MEM` memory error
- `ESP_ERR_INVALID_ARG` parameter error

#### Parameters

- `host_name`: host name to look for
- `timeout`: time in milliseconds to wait for answer. If 0, `max_results` needs to be defined
- `addr`: pointer to the resulting IP6 address

`esp_err_t mdns_handle_system_event (void *ctx, system_event_t *event)`

System event handler This method controls the service state on all active interfaces and applications are required to call it from the system event handler for normal operation of mDNS service.

#### Parameters

- `ctx`: The system event context
- `event`: The system event

## Structures

**struct mdns\_txt\_item\_t**  
mDNS basic text item structure Used in mdns\_service\_add()

### Public Members

char **\*key**  
item key name

char **\*value**  
item value string

**struct mdns\_ip\_addr\_s**  
mDNS query linked list IP item

### Public Members

ip\_addr\_t **addr**  
IP address

**struct mdns\_ip\_addr\_s \*next**  
next IP, or NULL for the last IP in the list

**struct mdns\_result\_s**  
mDNS query result structure

### Public Members

**struct mdns\_result\_s \*next**  
next result, or NULL for the last result in the list

tcpip\_adapter\_if\_t **tcpip\_if**  
interface on which the result came (AP/STA/ETH)

*mdns\_ip\_protocol\_t* **ip\_protocol**  
ip\_protocol type of the interface (v4/v6)

char **\*instance\_name**  
instance name

char **\*hostname**  
hostname

uint16\_t **port**  
service port

*mdns\_txt\_item\_t* **\*txt**  
txt record

size\_t **txt\_count**  
number of txt items

*mdns\_ip\_addr\_t* **\*addr**  
linked list of IP addresses found

## Macros

`MDNS_TYPE_A`  
`MDNS_TYPE_PTR`  
`MDNS_TYPE_TXT`  
`MDNS_TYPE_AAAA`  
`MDNS_TYPE_SRV`  
`MDNS_TYPE_OPT`  
`MDNS_TYPE_NSEC`  
`MDNS_TYPE_ANY`

## Type Definitions

```
typedef struct mdns_ip_addr_s mdns_ip_addr_t  
    mDNS query linked list IP item  
  
typedef struct mdns_result_s mdns_result_t  
    mDNS query result structure
```

## Enumerations

```
enum mdns_ip_protocol_t  
    mDNS enum to specify the ip_protocol type  
  
    Values:  
  
    MDNS_IP_PROTOCOL_V4  
    MDNS_IP_PROTOCOL_V6  
    MDNS_IP_PROTOCOL_MAX
```

Example code for this API section is provided in `protocols` directory of ESP-IDF examples.

## 2.6 Storage API

### 2.6.1 SPI Flash APIs

#### Overview

The `spi_flash` component contains APIs related to reading, writing, erasing, memory mapping data in the external SPI flash. It also has higher-level APIs which work with partitions defined in the *partition table*.

Note that all the functionality is limited to the “main” SPI flash chip, the same SPI flash chip from which program runs. For `spi_flash_*` functions, this is a software limitation. The underlying ROM functions which work with SPI flash do not have provisions for working with flash chips attached to SPI peripherals other than SPI0.

## SPI flash access APIs

This is the set of APIs for working with data in flash:

- `spi_flash_read` used to read data from flash to RAM
- `spi_flash_write` used to write data from RAM to flash
- `spi_flash_erase_sector` used to erase individual sectors of flash
- `spi_flash_erase_range` used to erase range of addresses in flash
- `spi_flash_get_chip_size` returns flash chip size, in bytes, as configured in `menuconfig`

Generally, try to avoid using the raw SPI flash functions 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 disabling detection in `make menuconfig` (under Serial Flasher Config).

If it is necessary to override the configured flash size at runtime, it is possible to set the `chip_size` member of `g_rom_flashchip` structure. This size is used by `spi_flash_*` functions (in both software & ROM) for bounds checking.

## Concurrency Constraints

Because the SPI flash is also used for firmware execution (via the instruction & data caches), these caches must be disabled while reading/writing/erasing. This means that both CPUs must be running code from IRAM and only reading data from DRAM while flash write operations occur.

Refer to the [application memory layout](#) documentation for an explanation of the differences between IRAM, DRAM and flash cache.

To avoid reading flash cache accidentally, when one CPU commences a flash write or erase operation the other CPU is put into a blocked state and all non-IRAM-safe interrupts are disabled on both CPUs, until the flash operation completes.

## IRAM-Safe Interrupt Handlers

If you have an interrupt handler that you want to execute even when a flash operation is in progress (for example, for low latency operations), set the `ESP_INTR_FLAG_IRAM` flag when the *interrupt handler is registered*.

You must ensure all data and functions accessed by these interrupt handlers are located in IRAM or DRAM. This includes any functions that the handler calls.

Use the `IRAM_ATTR` attribute for functions:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

Use the `DRAM_ATTR` and `DRAM_STR` attributes for constant data:

```
void IRAM_ATTR gpio_isr_handler(void* arg)
{
    const static DRAM_ATTR uint8_t INDEX_DATA[] = { 45, 33, 12, 0 };
    const static char *MSG = DRAM_STR("I am a string stored in RAM");
}
```

Note that knowing which data should be marked with `DRAM_ATTR` can be hard, the compiler will sometimes recognise that a variable or expression is constant (even if it is not marked `const`) and optimise it into flash, unless it is marked with `DRAM_ATTR`.

If a function or symbol is not correctly put into IRAM/DRAM and the interrupt handler reads from the flash cache during a flash operation, it will cause a crash due to Illegal Instruction exception (for code which should be in IRAM) or garbage data to be read (for constant data which should be in DRAM).

### Partition table APIs

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 about partition tables can be found [here](#).

This component provides APIs to enumerate partitions found in the partition table and perform operations on them. These functions are declared in `esp_partition.h`:

- `esp_partition_find` used to search partition table for entries with specific type, returns an opaque iterator
- `esp_partition_get` returns a structure describing the partition, for the given iterator
- `esp_partition_next` advances iterator to the next partition found
- `esp_partition_iterator_release` releases iterator returned by `esp_partition_find`
- `esp_partition_find_first` is a convenience function which returns structure describing the first partition found by `esp_partition_find`
- `esp_partition_read`, `esp_partition_write`, `esp_partition_erase_range` are equivalent to `spi_flash_read`, `spi_flash_write`, `spi_flash_erase_range`, but operate within partition boundaries

Most application code should use `esp_partition_*` APIs instead of lower level `spi_flash_*` APIs. Partition APIs do bounds checking and calculate correct offsets in flash based on data stored in partition table.

### SPI Flash Encryption

It is possible to encrypt SPI flash contents, and have it transparently decrypted by hardware.

Refer to the [Flash Encryption documentation](#) for more details.

### Memory mapping APIs

ESP32 features memory hardware which allows regions of flash memory to be mapped into instruction and data address spaces. This mapping works only for read operations, it is not possible to modify contents of flash memory by writing to mapped memory region. Mapping happens in 64KB pages. Memory mapping hardware can map up to 4 megabytes of flash into data address space, and up to 16 megabytes of flash into instruction address space. See the technical reference manual for more details about memory mapping hardware.

Note that some number of 64KB pages is used to map the application itself into memory, so the actual number of available 64KB pages may be less.

Reading data from flash using a memory mapped region is the only way to decrypt contents of flash when *flash encryption* is enabled. Decryption is performed at hardware level.

Memory mapping APIs are declared in `esp_spi_flash.h` and `esp_partition.h`:

- `spi_flash_mmap` maps a region of physical flash addresses into instruction space or data space of the CPU
- `spi_flash_munmap` unmaps previously mapped region
- `esp_partition_mmap` maps part of a partition into the instruction space or data space of the CPU

Differences between `spi_flash_mmap` and `esp_partition_mmap` are as follows:

- `spi_flash_mmap` must be given a 64KB aligned physical address
- `esp_partition_mmap` may be given an arbitrary offset within the partition, it will adjust returned pointer to mapped memory as necessary

Note that because memory mapping happens in 64KB blocks, it may be possible to read data outside of the partition provided to `esp_partition_mmap`.

### See also

- [Partition Table documentation](#)
- [Over The Air Update \(OTA\) API](#) provides high-level API for updating app firmware stored in flash.
- [Non-Volatile Storage \(NVS\) API](#) provides a structured API for storing small items of data in SPI flash.

### Implementation details

In order to perform some flash operations, we need to make sure both CPUs are not running any code from flash for the duration of the flash operation. In a single-core setup this is easy: we disable interrupts/scheduler and do the flash operation. In the dual-core setup this is slightly more complicated. We need to make sure that the other CPU doesn't run any code from flash.

When SPI flash API is called on CPU A (can be PRO or APP), we start `spi_flash_op_block_func` function on CPU B using `esp_ipc_call` API. This API wakes up high priority task on CPU B and tells it to execute given function, in this case `spi_flash_op_block_func`. This function disables cache on CPU B and signals that cache is disabled by setting `s_flash_op_can_start` flag. Then the task on CPU A disables cache as well, and proceeds to execute flash operation.

While flash operation is running, interrupts can still run on CPUs A and B. We assume that all interrupt code is placed into RAM. Once interrupt allocation API is added, we should add a flag to request interrupt to be disabled for the duration of flash operations.

Once flash operation is complete, 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), we simply disable both caches, no inter-CPU communication takes place.

### API Reference - SPI Flash

#### Header File

- `spi_flash/include/esp_spi_flash.h`

## Functions

void **spi\_flash\_init** ()

Initialize SPI flash access driver.

This function must be called exactly once, before any other `spi_flash_*` functions are called. Currently this function is called from startup code. There is no need to call it from application code.

size\_t **spi\_flash\_get\_chip\_size** ()

Get flash chip size, as set in binary image header.

**Note** This value does not necessarily match real flash size.

**Return** size of flash chip, in bytes

esp\_err\_t **spi\_flash\_erase\_sector** (size\_t *sector*)

Erase the Flash sector.

**Return** esp\_err\_t

### Parameters

- *sector*: Sector number, the count starts at sector 0, 4KB per sector.

esp\_err\_t **spi\_flash\_erase\_range** (size\_t *start\_address*, size\_t *size*)

Erase a range of flash sectors.

**Return** esp\_err\_t

### Parameters

- *start\_address*: Address where erase operation has to start. Must be 4kB-aligned
- *size*: Size of erased range, in bytes. Must be divisible by 4kB.

esp\_err\_t **spi\_flash\_write** (size\_t *dest\_addr*, const void \**src*, size\_t *size*)

Write data to Flash.

**Note** If source address is in DROM, this function will return `ESP_ERR_INVALID_ARG`.

**Return** esp\_err\_t

### Parameters

- *dest\_addr*: destination address in Flash. Must be a multiple of 4 bytes.
- *src*: pointer to the source buffer.
- *size*: length of data, in bytes. Must be a multiple of 4 bytes.

esp\_err\_t **spi\_flash\_write\_encrypted** (size\_t *dest\_addr*, const void \**src*, size\_t *size*)

Write data encrypted to Flash.

**Note** Flash encryption must be enabled for this function to work.

**Note** Flash encryption must be enabled when calling this function. If flash encryption is disabled, the function returns `ESP_ERR_INVALID_STATE`. Use `esp_flash_encryption_enabled()` function to determine if flash encryption is enabled.

**Note** Both *dest\_addr* and *size* must be multiples of 16 bytes. For absolute best performance, both *dest\_addr* and *size* arguments should be multiples of 32 bytes.

**Return** `esp_err_t`

**Parameters**

- `dest_addr`: destination address in Flash. Must be a multiple of 16 bytes.
- `src`: pointer to the source buffer.
- `size`: length of data, in bytes. Must be a multiple of 16 bytes.

`esp_err_t spi_flash_read` (`size_t src_addr`, `void *dest`, `size_t size`)  
Read data from Flash.

**Return** `esp_err_t`

**Parameters**

- `src_addr`: source address of the data in Flash.
- `dest`: pointer to the destination buffer
- `size`: length of data

`esp_err_t spi_flash_read_encrypted` (`size_t src`, `void *dest`, `size_t size`)  
Read data from Encrypted Flash.

If flash encryption is enabled, this function will transparently decrypt data as it is read. If flash encryption is not enabled, this function behaves the same as `spi_flash_read()`.

See `spi_flash_encryption_enabled()` for a function to check if flash encryption is enabled.

**Return** `esp_err_t`

**Parameters**

- `src`: source address of the data in Flash.
- `dest`: pointer to the destination buffer
- `size`: length of data

`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 64k MMU pages and configures them to map request region of flash memory into data address space or into instruction address space. It may reuse MMU pages which already provide required mapping. As with any allocator, there is possibility of fragmentation of address space if `mmap/munmap` are heavily used. To troubleshoot issues with page allocation, use `spi_flash_mmap_dump` function.

**Return** `ESP_OK` on success, `ESP_ERR_NO_MEM` if pages can not be allocated

**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 which has to be mapped. This size will be rounded up to a 64k boundary.
- `memory`: Memory space where the region should be mapped
- `out_ptr`: Output, pointer to the mapped memory region
- `out_handle`: Output, handle which should be used for `spi_flash_munmap` call

esp\_err\_t **spi\_flash\_mmap\_pages** (int \*pages, size\_t pagecount, 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 64k MMU pages and configures them to map the indicated pages of flash memory contiguously into data address space or into instruction 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.

**Return** ESP\_OK on success, ESP\_ERR\_NO\_MEM if pages can not be allocated

#### Parameters

- pages: An array of numbers indicating the 64K pages in flash to be mapped contiguously into memory. These indicate the indexes of the 64K pages, not the byte-size addresses as used in other functions.
- pagecount: Size of the pages array
- memory: Memory space where the region should be mapped
- out\_ptr: Output, pointer to the mapped memory region
- out\_handle: Output, handle which should be used for spi\_flash\_munmap call

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 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

void **spi\_flash\_mmap\_dump** ()

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.

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 free page number of the mmu table which can mmap, when you want to call spi\_flash\_mmap to mmap an ranger of flash data to Dcache or Icache memmory region, maybe the size of MMU table will exceed,so if you are not sure the size need mmap is ok, can call the interface and watch how many MMU table page can be mmaped.

**Return** number of free pages which can be mmaped

#### Parameters

- memory: memmory type of MMU table free page

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 flash map space can be looked up.

**Return**

- `SPI_FLASH_CACHE2PHYS_FAIL` If cache address is outside flash cache region, or the address is not mapped.
- Otherwise, returns physical offset in flash

**Parameters**

- `cached`: Pointer to flashed cached memory.

**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.

**Return**

- 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`.

**Parameters**

- `phys_offs`: Physical offset in flash memory to look up.
- `memory`: Memory type to look up a flash cache address mapping for (IROM or DROM)

bool **spi\_flash\_cache\_enabled** ()

Check at runtime if flash cache is enabled on both CPUs.

**Return** true if both CPUs have flash cache enabled, false otherwise.

void **spi\_flash\_guard\_set** (const *spi\_flash\_guard\_funcs\_t* \**funcs*)

Sets guard functions to access flash.

**Note** Pointed structure and corresponding guard functions should not reside in flash. For example structure can be placed in DRAM and functions in IRAM sections.

**Parameters**

- `funcs`: pointer to structure holding flash access guard functions.

const *spi\_flash\_guard\_funcs\_t* \***spi\_flash\_guard\_get** ()

Get the guard functions used for flash access.

**Return** The guard functions that were set via `spi_flash_guard_set()`. These functions can be called if implementing custom low-level SPI flash operations.

## Structures

### **struct spi\_flash\_guard\_funcs\_t**

Structure holding SPI flash access critical sections management functions.

Flash API uses two types of flash access management functions: 1) Functions which prepare/restore flash cache and interrupts before calling appropriate ROM functions (SPIWrite, SPIRead and SPIEraseBlock):

- 'start' function should disables flash cache and non-IRAM interrupts and is invoked before the call to one of ROM function above.
- 'end' function should restore state of flash cache and non-IRAM interrupts and is invoked after the call to one of ROM function above. These two functions are not recursive. 2) Functions which synchronizes access to internal data used by flash API. This functions are mostly intended to synchronize access to flash API internal data in multithreaded environment and use OS primitives:
- 'op\_lock' locks access to flash API internal data.
- 'op\_unlock' unlocks access to flash API internal data. These two functions are recursive and can be used around the outside of multiple calls to 'start' & 'end', in order to create atomic multi-part flash operations.

Different versions of the guarding functions should be used depending on the context of execution (with or without functional OS). In normal conditions when flash API is called from task the functions use OS primitives. When there is no OS at all or when it is not guaranteed that OS is functional (accessing flash from exception handler) these functions cannot use OS primitives or even does not need them (multithreaded access is not possible).

**Note** Structure and corresponding guard functions should not reside in flash. For example structure can be placed in DRAM and functions in IRAM sections.

## Public Members

*spi\_flash\_guard\_start\_func\_t* **start**  
critical section start function.

*spi\_flash\_guard\_end\_func\_t* **end**  
critical section end function.

*spi\_flash\_op\_lock\_func\_t* **op\_lock**  
flash access API lock function.

*spi\_flash\_op\_unlock\_func\_t* **op\_unlock**  
flash access API unlock function.

## Macros

**ESP\_ERR\_FLASH\_BASE**

**ESP\_ERR\_FLASH\_OP\_FAIL**

**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.

**typedef** void (\***spi\_flash\_guard\_start\_func\_t**) (void)  
 SPI flash critical section enter function.

**typedef** void (\***spi\_flash\_guard\_end\_func\_t**) (void)  
 SPI flash critical section exit function.

**typedef** void (\***spi\_flash\_op\_lock\_func\_t**) (void)  
 SPI flash operation lock function.

**typedef** void (\***spi\_flash\_op\_unlock\_func\_t**) (void)  
 SPI flash operation unlock function.

## Enumerations

**enum** **spi\_flash\_mmap\_memory\_t**  
 Enumeration which specifies memory space requested in an mmap call.

*Values:*

**SPI\_FLASH\_MMAP\_DATA**  
 map to data memory (Vaddr0), allows byte-aligned access, 4 MB total

**SPI\_FLASH\_MMAP\_INST**  
 map to instruction memory (Vaddr1-3), allows only 4-byte-aligned access, 11 MB total

## API Reference - Partition Table

### Header File

- spi\_flash/include/esp\_partition.h

### Functions

*esp\_partition\_iterator\_t* **esp\_partition\_find**(*esp\_partition\_type\_t* type, *esp\_partition\_subtype\_t* subtype, **const** char \*label)

Find partition based on one or more parameters.

**Return** iterator which can be used to enumerate all the partitions found, or NULL if no partitions were found. Iterator obtained through this function has to be released using `esp_partition_iterator_release` when not used any more.

#### Parameters

- type: Partition type, one of `esp_partition_type_t` values
- subtype: Partition subtype, one of `esp_partition_subtype_t` values. To find all partitions of given type, use `ESP_PARTITION_SUBTYPE_ANY`.
- label: (optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

```
const esp_partition_t *esp_partition_find_first (esp_partition_type_t      type,  
                                             esp_partition_subtype_t  subtype, const  
                                             char *label)
```

Find first partition based on one or more parameters.

**Return** pointer to *esp\_partition\_t* structure, or NULL if no partition is found. This pointer is valid for the lifetime of the application.

#### Parameters

- *type*: Partition type, one of *esp\_partition\_type\_t* values
- *subtype*: Partition subtype, one of *esp\_partition\_subtype\_t* values. To find all partitions of given type, use `ESP_PARTITION_SUBTYPE_ANY`.
- *label*: (optional) Partition label. Set this value if looking for partition with a specific name. Pass NULL otherwise.

```
const esp_partition_t *esp_partition_get (esp_partition_iterator_t iterator)
```

Get *esp\_partition\_t* structure for given partition.

**Return** pointer to *esp\_partition\_t* structure. This pointer is valid for the lifetime of the application.

#### Parameters

- *iterator*: Iterator obtained using `esp_partition_find`. Must be non-NULL.

```
esp_partition_iterator_t esp_partition_next (esp_partition_iterator_t iterator)
```

Move partition iterator to the next partition found.

Any copies of the iterator will be invalid after this call.

**Return** NULL if no partition was found, valid *esp\_partition\_iterator\_t* otherwise.

#### Parameters

- *iterator*: Iterator obtained using `esp_partition_find`. Must be non-NULL.

```
void esp_partition_iterator_release (esp_partition_iterator_t iterator)
```

Release partition iterator.

#### Parameters

- *iterator*: Iterator obtained using `esp_partition_find`. Must be non-NULL.

```
const esp_partition_t *esp_partition_verify (const esp_partition_t *partition)
```

Verify partition data.

Given a pointer to partition data, verify this partition exists in the partition table (all fields match.)

This function is also useful to take partition data which may be in a RAM buffer and convert it to a pointer to the permanent partition data stored in flash.

Pointers returned from this function can be compared directly to the address of any pointer returned from `esp_partition_get()`, as a test for equality.

#### Return

- If partition not found, returns NULL.
- If found, returns a pointer to the *esp\_partition\_t* structure in flash. This pointer is always valid for the lifetime of the application.

**Parameters**

- `partition`: Pointer to partition data to verify. Must be non-NULL. All fields of this structure must match the partition table entry in flash for this function to return a successful match.

`esp_err_t esp_partition_read(const esp_partition_t *partition, size_t src_offset, void *dst, size_t size)`

Read data from the partition.

**Return** ESP\_OK, if data was read successfully; ESP\_ERR\_INVALID\_ARG, if `src_offset` exceeds partition size; ESP\_ERR\_INVALID\_SIZE, if read would go out of bounds of the partition; or one of error codes from lower-level flash driver.

**Parameters**

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `dst`: Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `src_offset`: Address of the data to be read, relative to the beginning of the partition.
- `size`: Size of data to be read, in bytes.

`esp_err_t esp_partition_write(const esp_partition_t *partition, size_t dst_offset, const void *src, size_t size)`

Write data to the partition.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using `esp_partition_erase_range` function.

Partitions marked with an encryption flag will automatically be written via the `spi_flash_write_encrypted()` function. If writing to an encrypted partition, all write offsets and lengths must be multiples of 16 bytes. See the `spi_flash_write_encrypted()` function for more details. Unencrypted partitions do not have this restriction.

**Note** Prior to writing to flash memory, make sure it has been erased with `esp_partition_erase_range` call.

**Return** ESP\_OK, if data was written successfully; ESP\_ERR\_INVALID\_ARG, if `dst_offset` exceeds partition size; ESP\_ERR\_INVALID\_SIZE, if write would go out of bounds of the partition; or one of error codes from lower-level flash driver.

**Parameters**

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `dst_offset`: Address where the data should be written, relative to the beginning of the partition.
- `src`: Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `size`: Size of data to be written, in bytes.

`esp_err_t esp_partition_erase_range(const esp_partition_t *partition, uint32_t start_addr, uint32_t size)`

Erase part of the partition.

**Return** ESP\_OK, if the range was erased successfully; ESP\_ERR\_INVALID\_ARG, if iterator or `dst` are NULL; ESP\_ERR\_INVALID\_SIZE, if erase would go out of bounds of the partition; or one of error codes from lower-level flash driver.

### Parameters

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `start_addr`: Address where erase operation should start. Must be aligned to 4 kilobytes.
- `size`: Size of the range which should be erased, in bytes. Must be divisible by 4 kilobytes.

```
esp_err_t esp_partition_mmap(const esp_partition_t *partition, uint32_t offset, uint32_t size,
                            spi_flash_mmap_memory_t memory, const void **out_ptr,
                            spi_flash_mmap_handle_t *out_handle)
```

Configure MMU to map partition into data memory.

Unlike `spi_flash_mmap` function, which requires a 64kB aligned base address, this function doesn't impose such a requirement. If offset results in a flash address which is not aligned to 64kB boundary, address will be rounded to the lower 64kB boundary, so that mapped region includes requested range. Pointer returned via `out_ptr` argument will be adjusted to point to the requested offset (not necessarily to the beginning of mmap-ed region).

To release mapped memory, pass handle returned via `out_handle` argument to `spi_flash_munmap` function.

**Return** `ESP_OK`, if successful

### Parameters

- `partition`: Pointer to partition structure obtained using `esp_partition_find_first` or `esp_partition_get`. Must be non-NULL.
- `offset`: Offset from the beginning of partition where mapping should start.
- `size`: Size of the area to be mapped.
- `memory`: Memory space where the region should be mapped
- `out_ptr`: Output, pointer to the mapped memory region
- `out_handle`: Output, handle which should be used for `spi_flash_munmap` call

## Structures

```
struct esp_partition_t
```

partition information structure

This is not the format in flash, that format is `esp_partition_info_t`.

However, this is the format used by this API.

### Public Members

```
esp_partition_type_t type
```

partition type (app/data)

```
esp_partition_subtype_t subtype
```

partition subtype

```
uint32_t address
```

starting address of the partition in flash

```
uint32_t size
```

size of the partition, in bytes

char **label**[17]  
 partition label, zero-terminated ASCII string

bool **encrypted**  
 flag is set to true if partition is encrypted

## Macros

**ESP\_PARTITION\_SUBTYPE\_OTA**(i)  
 Convenience macro to get esp\_partition\_subtype\_t value for the i-th OTA partition.

## Type Definitions

**typedef struct** esp\_partition\_iterator\_opaque\_ \***esp\_partition\_iterator\_t**  
 Opaque partition iterator type.

## Enumerations

**enum esp\_partition\_type\_t**  
 Partition type.

**Note** Keep this enum in sync with PartitionDefinition class gen\_esp32part.py

*Values:*

**ESP\_PARTITION\_TYPE\_APP** = 0x00  
 Application partition type.

**ESP\_PARTITION\_TYPE\_DATA** = 0x01  
 Data partition type.

**enum esp\_partition\_subtype\_t**  
 Partition subtype.

**Note** Keep this enum in sync with PartitionDefinition class gen\_esp32part.py

*Values:*

**ESP\_PARTITION\_SUBTYPE\_APP\_FACTORY** = 0x00  
 Factory application partition.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN** = 0x10  
 Base for OTA partition subtypes.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_0** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 0  
 OTA partition 0.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_1** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 1  
 OTA partition 1.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_2** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 2  
 OTA partition 2.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_3** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 3  
 OTA partition 3.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_4** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 4  
OTA partition 4.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_5** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 5  
OTA partition 5.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_6** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 6  
OTA partition 6.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_7** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 7  
OTA partition 7.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_8** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 8  
OTA partition 8.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_9** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 9  
OTA partition 9.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_10** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 10  
OTA partition 10.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_11** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 11  
OTA partition 11.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_12** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 12  
OTA partition 12.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_13** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 13  
OTA partition 13.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_14** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 14  
OTA partition 14.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_15** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 15  
OTA partition 15.

**ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MAX** = ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN + 16  
Max subtype of OTA partition.

**ESP\_PARTITION\_SUBTYPE\_APP\_TEST** = 0x20  
Test application partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_OTA** = 0x00  
OTA selection partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_PHY** = 0x01  
PHY init data partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_NVS** = 0x02  
NVS partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_COREDUMP** = 0x03  
COREDUMP partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_ESPHTTPD** = 0x80  
ESPHTTPD partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_FAT** = 0x81  
FAT partition.

**ESP\_PARTITION\_SUBTYPE\_DATA\_SPIFFS** = 0x82  
SPIFFS partition.

**ESP\_PARTITION\_SUBTYPE\_ANY** = 0xff

Used to search for partitions with any subtype.

## API Reference - Flash Encrypt

### Header File

- `bootloader_support/include/esp_flash_encrypt.h`

### Functions

**static bool** `esp_flash_encryption_enabled` (void)

Is flash encryption currently enabled in hardware?

Flash encryption is enabled if the FLASH\_CRYPT\_CNT efuse has an odd number of bits set.

**Return** true if flash encryption is enabled.

`esp_err_t` `esp_flash_encrypt_check_and_update` (void)

`esp_err_t` `esp_flash_encrypt_region` (uint32\_t *src\_addr*, size\_t *data\_length*)

Encrypt-in-place a block of flash sectors.

**Return** ESP\_OK if all operations succeeded, ESP\_ERR\_FLASH\_OP\_FAIL if SPI flash fails, ESP\_ERR\_FLASH\_OP\_TIMEOUT if flash times out.

#### Parameters

- *src\_addr*: Source offset in flash. Should be multiple of 4096 bytes.
- *data\_length*: Length of data to encrypt in bytes. Will be rounded up to next multiple of 4096 bytes.

void `esp_flash_write_protect_crypt_cnt` ()

Write protect FLASH\_CRYPT\_CNT.

Intended to be called as a part of boot process if flash encryption should be permanently enabled. This should protect against serial re-flashing of an unauthorised code in absence of secure boot or if secure boot protection is bypassed.

**Return**

## 2.6.2 Non-volatile storage library

### Introduction

Non-volatile storage (NVS) library is designed to store key-value pairs in flash. This sections introduces some concepts used by NVS.

## Underlying storage

Currently NVS uses a portion of main flash memory through `spi_flash_{read|write|erase}` APIs. The library uses the all the partitions with `data` type and `nvs` subtype. The application can choose to use the partition with label `nvs` through `nvs_open` API or any of the other partition by specifying its name through `nvs_open_from_part` API.

Future versions of this library may add 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 `make 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 type ‘string’ and ‘blob’. If storing large blobs or strings is required, 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, 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 and blob values are currently limited to 1984 bytes. For strings, this includes the null terminator.

---

Additional types, such as `float` and `double` may be added later.

Keys are required to be unique. Writing a value for a key which already exists behaves as follows:

- if the new value is of the same type as old one, value is updated
- if the new value has different data type, an error is returned

Data type check is also performed when reading a value. An error is returned if data type of read operation doesn't match the data type of the value.

## Namespaces

To mitigate potential conflicts in key names between different components, NVS assigns each key-value pair to one of namespaces. Namespace names follow the same rules as key names, i.e. 15 character maximum length. Namespace name is specified in the `nvs_open` or `nvs_open_from_part` call. This call returns an opaque handle, which is used in subsequent calls to `nvs_read_*`, `nvs_write_*`, and `nvs_commit` functions. This way, handle is associated with a namespace, and key names will not collide with same names in other namespaces. Please note that the namespaces with same name in different NVS partitions are considered as separate namespaces.

## Security, tampering, and robustness

NVS library doesn't implement tamper prevention measures. It is possible for anyone with physical access to the flash chip to alter, erase, or add key-value pairs.

NVS is compatible with the ESP32 flash encryption system, and it can store key-value pairs in an encrypted form. Some metadata, like page state and write/erase flags of individual entries can not be encrypted as they are represented as bits of flash memory for efficient access and manipulation. Flash encryption can prevent some forms of modification:

- replacing keys or values with arbitrary data
- changing data types of values

The following forms of modification are still possible when flash encryption is used:

- erasing a page completely, removing all key-value pairs which were stored in that page
- corrupting data in a page, which will cause the page to be erased automatically when such condition is detected
- rolling back the contents of flash memory to an earlier snapshot
- merging two snapshots of flash memory, rolling back some key-value pairs to an earlier state (although this is possible to mitigate with the current design — TODO)

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, expect for the new key-value pair if it was being written at the moment of power off. The library should also be able to initialize properly with any random data present in flash memory.

## Internals

### Log of key-value pairs

NVS stores key-value pairs sequentially, with new key-value pairs being added at the end. When a value of any given key has to be updated, new key-value pair is added at the end of the log and 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 isn't used to store any data at this point and doesn't 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. At most 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 when any API call returns. In case of a sudden power off, move-and-erase process will be completed upon next power on.



## Entry and entry state bitmap

Each entry can be in one of the following three states. Each state is represented with two bits in the entry state bitmap. Final four bits in the bitmap (256 - 2 \* 126) are unused.

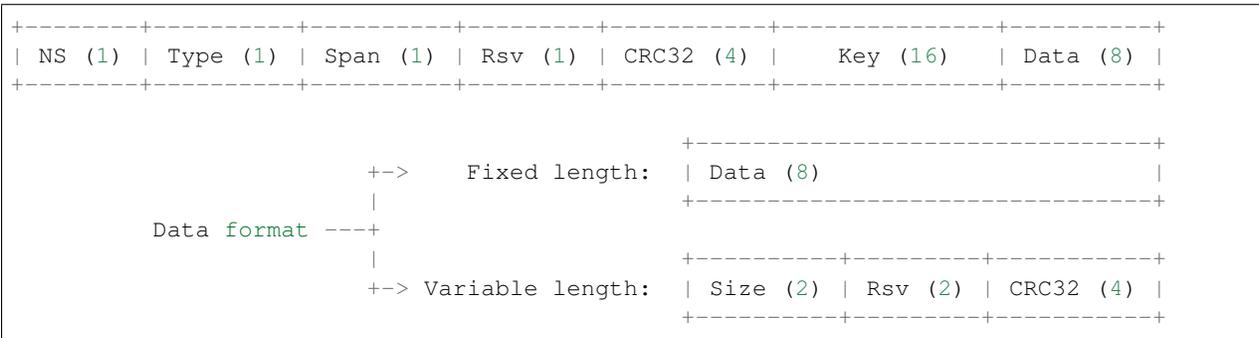
**Empty (2'b11)** Nothing is written into the specific entry yet. It is in an uninitialized state (all bytes 0xff).

**Written (2'b10)** A key-value pair (or part of key-value pair which spans multiple entries) has been written into the entry.

**Erased (2'b00)** A key-value pair in this entry has been discarded. Contents of this entry will not be parsed anymore.

## Structure of entry

For values of primitive types (currently integers from 1 to 8 bytes long), entry holds one key-value pair. For string and blob types, entry holds part of the whole key-value pair. In case when a key-value pair spans multiple entries, all entries are stored in the same page.



Individual fields in entry structure have the following meanings:

**NS** Namespace index for this entry. See section on namespaces implementation for explanation of this value.

**Type** One byte indicating data type of value. See `ItemType` enumeration in `nvs_types.h` 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.

**Rsv** Unused field, 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 key name. Maximum string length is 15 bytes, excluding 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 string and blob values, these 8 bytes hold additional data about the value, described next:

**Size** (Only for strings and blobs.) Size, in bytes, of actual data. For strings, this includes zero terminator.

**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. *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. Namespaces identifiers (strings) are stored as keys of key-value pairs in namespace with index 0. Values corresponding to these keys are indexes of these namespaces.

NS=0 Type=uint8_t Key="wifi" Value=1	Entry describing namespace "wifi"
NS=1 Type=uint32_t Key="channel" Value=6	Key "channel" in namespace "wifi"
NS=0 Type=uint8_t Key="pwm" Value=2	Entry describing namespace "pwm"
NS=2 Type=uint16_t Key="channel" Value=20	Key "channel" in namespace "pwm"

## Item hash list

To reduce the number of reads performed from flash memory, each member of 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 search for item hash in the hash list. This gives the item index within the page, if such an item exists. Due to a hash collision it is possible that a different item will be found. This is handled by falling back to iteration over items in flash.

Each node in hash list contains a 24-bit hash and 8-bit item index. Hash is calculated based on item namespace and key name. CRC32 is used for calculation, result is truncated to 24 bits. To reduce overhead of storing 32-bit entries in a linked list, list is implemented as a doubly-linked list of arrays. Each array holds 29 entries, for the total size of 128 bytes, together with linked list pointers and 32-bit count field. Minimal amount of extra RAM usage per page is therefore 128 bytes, maximum is 640 bytes.

## Application Example

Two examples are provided in `storage` directory of ESP-IDF examples:

### `storage/nvs_rw_value`

Demonstrates how to read and write a single integer value using NVS.

The value holds the number of ESP32 module restarts. Since it is written to NVS, the value is preserved between restarts.

Example also shows how to check if read / write operation was successful, or certain value is not initialized in NVS. Diagnostic is provided in plain text to help track program flow and capture any issues on the way.

### `storage/nvs_rw_blob`

Demonstrates how to read and write a single integer value and a blob (binary large object) using NVS to preserve them between ESP32 module restarts.

- value - tracks number of ESP32 module soft and hard restarts.
- blob - contains a table with module run times. The table is read from NVS to dynamically allocated RAM. New run time is added to the table on each manually triggered soft restart and written back to NVS. Triggering is done by pulling down GPIO0.

Example also shows how to implement diagnostics if read / write operation was successful.

## API Reference

### Header File

- `nvs_flash/include/nvs_flash.h`

### Functions

`esp_err_t nvs_flash_init` (void)

Initialize the default NVS partition.

This API initialises the default NVS partition. The default NVS partition is the one that is labeled “nvs” in the partition table.

#### Return

- `ESP_OK` if storage was successfully initialized.
- `ESP_ERR_NVS_NO_FREE_PAGES` if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- `ESP_ERR_NOT_FOUND` if no partition with label “nvs” is found in the partition table
- one of the error codes from the underlying flash storage driver

`esp_err_t nvs_flash_init_partition` (const char \**partition\_label*)

Initialize NVS flash storage for the specified partition.

#### Return

- `ESP_OK` if storage was successfully initialized.
- `ESP_ERR_NVS_NO_FREE_PAGES` if the NVS storage contains no empty pages (which may happen if NVS partition was truncated)
- `ESP_ERR_NOT_FOUND` if specified partition is not found in the partition table
- one of the error codes from the underlying flash storage driver

#### Parameters

- `partition_label`: Label of the partition. Note that internally a reference to passed value is kept and it should be accessible for future operations

`esp_err_t nvs_flash_deinit` (void)

Deinitialize NVS storage for the default NVS partition.

Default NVS partition is the partition with “nvs” label in the partition table.

#### Return

- `ESP_OK` on success (storage was deinitialized)
- `ESP_ERR_NVS_NOT_INITIALIZED` if the storage was not initialized prior to this call

`esp_err_t nvs_flash_deinit_partition` (const char \**partition\_label*)

Deinitialize NVS storage for the given NVS partition.

#### Return

- ESP\_OK on success
- ESP\_ERR\_NVS\_NOT\_INITIALIZED if the storage for given partition was not initialized prior to this call

#### Parameters

- `partition_label`: Label of the partition

`esp_err_t nvs_flash_erase` (void)

Erase the default NVS partition.

This function erases all contents of the default NVS partition (one with label “nvs”)

#### Return

- ESP\_OK on success
- ESP\_ERR\_NOT\_FOUND if there is no NVS partition labeled “nvs” in the partition table

`esp_err_t nvs_flash_erase_partition` (const char \**part\_name*)

Erase specified NVS partition.

This function erases all contents of specified NVS partition

#### Return

- ESP\_OK on success
- ESP\_ERR\_NOT\_FOUND if there is no NVS partition with the specified name in the partition table

#### Parameters

- `part_name`: Name (label) of the partition to be erased

### Header File

- [nvs\\_flash/include/nvs.h](#)

### Functions

`esp_err_t nvs_set_i8` (*nvs\_handle* handle, const char \**key*, int8\_t *value*)

set value for given key

This family of functions set value for the key, given its name. Note that actual storage will not be updated until `nvs_commit` function is called.

#### Return

- ESP\_OK if value was set successfully
- ESP\_ERR\_NVS\_INVALID\_HANDLE if handle has been closed or is NULL
- ESP\_ERR\_NVS\_READ\_ONLY if storage handle was opened as read only
- ESP\_ERR\_NVS\_INVALID\_NAME if key name doesn't satisfy constraints
- ESP\_ERR\_NVS\_NOT\_ENOUGH\_SPACE if there is not enough space in the underlying storage to save the value

- `ESP_ERR_NVS_REMOVE_FAILED` if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.
- `ESP_ERR_NVS_VALUE_TOO_LONG` if the string value is too long

### Parameters

- `handle`: Handle obtained from `nvs_open` function. Handles that were opened read only cannot be used.
- `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `value`: The value to set. For strings, the maximum length (including null character) is 1984 bytes.

```
esp_err_t nvs_set_u8(nvs_handle handle, const char *key, uint8_t value)
esp_err_t nvs_set_i16(nvs_handle handle, const char *key, int16_t value)
esp_err_t nvs_set_u16(nvs_handle handle, const char *key, uint16_t value)
esp_err_t nvs_set_i32(nvs_handle handle, const char *key, int32_t value)
esp_err_t nvs_set_u32(nvs_handle handle, const char *key, uint32_t value)
esp_err_t nvs_set_i64(nvs_handle handle, const char *key, int64_t value)
esp_err_t nvs_set_u64(nvs_handle handle, const char *key, uint64_t value)
esp_err_t nvs_set_str(nvs_handle handle, const char *key, const char *value)
esp_err_t nvs_get_i8(nvs_handle handle, const char *key, int8_t *out_value)
    get 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.

All functions expect `out_value` to be a pointer to an already allocated variable of the given type.

```
// Example of using nvs_get_i32:
int32_t max_buffer_size = 4096; // default value
esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
// if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
// have its default value.
```

### Return

- `ESP_OK` if the value was retrieved successfully
- `ESP_ERR_NVS_NOT_FOUND` if the requested key doesn't exist
- `ESP_ERR_NVS_INVALID_HANDLE` if handle has been closed or is `NULL`
- `ESP_ERR_NVS_INVALID_NAME` if key name doesn't satisfy constraints
- `ESP_ERR_NVS_INVALID_LENGTH` if length is not sufficient to store data

### Parameters

- `handle`: Handle obtained from `nvs_open` function.

- `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `out_value`: Pointer to the output value. May be NULL for `nvs_get_str` and `nvs_get_blob`, in this case required length will be returned in `length` argument.

`esp_err_t nvs_get_u8 (nvs_handle handle, const char *key, uint8_t *out_value)`

`esp_err_t nvs_get_i16 (nvs_handle handle, const char *key, int16_t *out_value)`

`esp_err_t nvs_get_u16 (nvs_handle handle, const char *key, uint16_t *out_value)`

`esp_err_t nvs_get_i32 (nvs_handle handle, const char *key, int32_t *out_value)`

`esp_err_t nvs_get_u32 (nvs_handle handle, const char *key, uint32_t *out_value)`

`esp_err_t nvs_get_i64 (nvs_handle handle, const char *key, int64_t *out_value)`

`esp_err_t nvs_get_u64 (nvs_handle handle, const char *key, uint64_t *out_value)`

`esp_err_t nvs_get_str (nvs_handle handle, const char *key, char *out_value, size_t *length)`  
get 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.

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);
```

### Return

- `ESP_OK` if the value was retrieved successfully
- `ESP_ERR_NVS_NOT_FOUND` if the requested key doesn't exist
- `ESP_ERR_NVS_INVALID_HANDLE` if `handle` has been closed or is NULL
- `ESP_ERR_NVS_INVALID_NAME` if key name doesn't satisfy constraints
- `ESP_ERR_NVS_INVALID_LENGTH` if `length` is not sufficient to store data

### Parameters

- `handle`: Handle obtained from `nvs_open` function.
- `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `out_value`: Pointer to the output value. May be `NULL` for `nvs_get_str` and `nvs_get_blob`, in this case required length will be returned in `length` argument.
- `length`: A non-zero pointer to the variable holding the length of `out_value`. In case `out_value` a zero, will be set to the length required to hold the value. In case `out_value` is not zero, will be set to the actual length of the value written. For `nvs_get_str` this includes zero terminator.

`esp_err_t nvs_get_blob` (*`nvs_handle` handle*, `const char *key`, `void *out_value`, `size_t *length`)

`esp_err_t nvs_open` (`const char *name`, *`nvs_open_mode` open\_mode*, *`nvs_handle` \*out\_handle*)

Open non-volatile storage with a given namespace from the default NVS partition.

Multiple internal ESP-IDF and third party application modules can store their key-value pairs in the NVS module. In order to reduce possible conflicts on key names, each module can use its own namespace. The default NVS partition is the one that is labelled "nvs" in the partition table.

### Return

- `ESP_OK` if storage handle was opened successfully
- `ESP_ERR_NVS_NOT_INITIALIZED` if the storage driver is not initialized
- `ESP_ERR_NVS_PART_NOT_FOUND` if the partition with label "nvs" is not found
- `ESP_ERR_NVS_NOT_FOUND` id namespace doesn't exist yet and mode is `NVS_READONLY`
- `ESP_ERR_NVS_INVALID_NAME` if namespace name doesn't satisfy constraints
- other error codes from the underlying storage driver

### Parameters

- `name`: Namespace name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `open_mode`: `NVS_READWRITE` or `NVS_READONLY`. If `NVS_READONLY`, will open a handle for reading only. All write requests will be rejected for this handle.
- `out_handle`: If successful (return code is zero), handle will be returned in this argument.

`esp_err_t nvs_open_from_partition` (`const char *part_name`, `const char *name`, *`nvs_open_mode` open\_mode*, *`nvs_handle` \*out\_handle*)

Open non-volatile storage with a given namespace from specified partition.

The behaviour is same as `nvs_open()` API. However this API can operate on a specified NVS partition instead of default NVS partition. Note that the specified partition must be registered with NVS using `nvs_flash_init_partition()` API.

### Return

- `ESP_OK` if storage handle was opened successfully
- `ESP_ERR_NVS_NOT_INITIALIZED` if the storage driver is not initialized
- `ESP_ERR_NVS_PART_NOT_FOUND` if the partition with specified name is not found
- `ESP_ERR_NVS_NOT_FOUND` id namespace doesn't exist yet and mode is `NVS_READONLY`
- `ESP_ERR_NVS_INVALID_NAME` if namespace name doesn't satisfy constraints

- other error codes from the underlying storage driver

#### Parameters

- `part_name`: Label (name) of the partition of interest for object read/write/erase
- `name`: Namespace name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.
- `open_mode`: NVS\_READWRITE or NVS\_READONLY. If NVS\_READONLY, will open a handle for reading only. All write requests will be rejected for this handle.
- `out_handle`: If successful (return code is zero), handle will be returned in this argument.

`esp_err_t nvs_set_blob` (*nvs\_handle* handle, **const** char \*key, **const** void \*value, size\_t length)  
set variable length binary value for given key

This family of functions set value for the key, given its name. Note that actual storage will not be updated until `nvs_commit` function is called.

#### Return

- ESP\_OK if value was set successfully
- ESP\_ERR\_NVS\_INVALID\_HANDLE if handle has been closed or is NULL
- ESP\_ERR\_NVS\_READ\_ONLY if storage handle was opened as read only
- ESP\_ERR\_NVS\_INVALID\_NAME if key name doesn't satisfy constraints
- ESP\_ERR\_NVS\_NOT\_ENOUGH\_SPACE if there is not enough space in the underlying storage to save the value
- ESP\_ERR\_NVS\_REMOVE\_FAILED if the value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.
- ESP\_ERR\_NVS\_VALUE\_TOO\_LONG if the value is too long

#### Parameters

- `handle`: Handle obtained from `nvs_open` function. Handles that were opened read only cannot be used.
- `key`: Key name. Maximal length is 15 characters. Shouldn't be empty.
- `value`: The value to set.
- `length`: length of binary value to set, in bytes; Maximum length is 1984 bytes.

`esp_err_t nvs_erase_key` (*nvs\_handle* handle, **const** char \*key)  
Erase key-value pair with given key name.

Note that actual storage may not be updated until `nvs_commit` function is called.

#### Return

- ESP\_OK if erase operation was successful
- ESP\_ERR\_NVS\_INVALID\_HANDLE if handle has been closed or is NULL
- ESP\_ERR\_NVS\_READ\_ONLY if handle was opened as read only
- ESP\_ERR\_NVS\_NOT\_FOUND if the requested key doesn't exist
- other error codes from the underlying storage driver

**Parameters**

- `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.
- `key`: Key name. Maximal length is determined by the underlying implementation, but is guaranteed to be at least 15 characters. Shouldn't be empty.

`esp_err_t nvs_erase_all` (*nvs\_handle* `handle`)

Erase all key-value pairs in a namespace.

Note that actual storage may not be updated until `nvs_commit` function is called.

**Return**

- `ESP_OK` if erase operation was successful
- `ESP_ERR_NVS_INVALID_HANDLE` if `handle` has been closed or is `NULL`
- `ESP_ERR_NVS_READ_ONLY` if `handle` was opened as read only
- other error codes from the underlying storage driver

**Parameters**

- `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.

`esp_err_t nvs_commit` (*nvs\_handle* `handle`)

Write any pending changes to non-volatile storage.

After setting any values, `nvs_commit()` must be called to ensure changes are written to non-volatile storage. Individual implementations may write to storage at other times, but this is not guaranteed.

**Return**

- `ESP_OK` if the changes have been written successfully
- `ESP_ERR_NVS_INVALID_HANDLE` if `handle` has been closed or is `NULL`
- other error codes from the underlying storage driver

**Parameters**

- `handle`: Storage handle obtained with `nvs_open`. Handles that were opened read only cannot be used.

`void nvs_close` (*nvs\_handle* `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`: Storage handle to close

## Macros

### **ESP\_ERR\_NVS\_BASE**

Starting number of error codes

### **ESP\_ERR\_NVS\_NOT\_INITIALIZED**

The storage driver is not initialized

### **ESP\_ERR\_NVS\_NOT\_FOUND**

Id namespace doesn't exist yet and mode is NVS\_READONLY

### **ESP\_ERR\_NVS\_TYPE\_MISMATCH**

The type of set or get operation doesn't match the type of value stored in NVS

### **ESP\_ERR\_NVS\_READ\_ONLY**

Storage handle was opened as read only

### **ESP\_ERR\_NVS\_NOT\_ENOUGH\_SPACE**

There is not enough space in the underlying storage to save the value

### **ESP\_ERR\_NVS\_INVALID\_NAME**

Namespace name doesn't satisfy constraints

### **ESP\_ERR\_NVS\_INVALID\_HANDLE**

Handle has been closed or is NULL

### **ESP\_ERR\_NVS\_REMOVE\_FAILED**

The value wasn't updated because flash write operation has failed. The value was written however, and update will be finished after re-initialization of nvs, provided that flash operation doesn't fail again.

### **ESP\_ERR\_NVS\_KEY\_TOO\_LONG**

Key name is too long

### **ESP\_ERR\_NVS\_PAGE\_FULL**

Internal error; never returned by nvs\_ API functions

### **ESP\_ERR\_NVS\_INVALID\_STATE**

NVS is in an inconsistent state due to a previous error. Call nvs\_flash\_init and nvs\_open again, then retry.

### **ESP\_ERR\_NVS\_INVALID\_LENGTH**

String or blob length is not sufficient to store data

### **ESP\_ERR\_NVS\_NO\_FREE\_PAGES**

NVS partition doesn't contain any empty pages. This may happen if NVS partition was truncated. Erase the whole partition and call nvs\_flash\_init again.

### **ESP\_ERR\_NVS\_VALUE\_TOO\_LONG**

String or blob length is longer than supported by the implementation

### **ESP\_ERR\_NVS\_PART\_NOT\_FOUND**

Partition with specified name is not found in the partition table

### **NVS\_DEFAULT\_PART\_NAME**

Default partition name of the NVS partition in the partition table

## Type Definitions

### **typedef uint32\_t nvs\_handle**

Opaque pointer type representing non-volatile storage handle

## Enumerations

### enum nvs\_open\_mode

Mode of opening the non-volatile storage.

*Values:*

#### NVS\_READONLY

Read only

#### NVS\_READWRITE

Read and write

## 2.6.3 Virtual filesystem component

### Overview

Virtual filesystem (VFS) component provides a unified interface for drivers which can perform operations on file-like objects. This can be a real filesystems (FAT, SPIFFS, etc.), or device drivers which exposes file-like interface.

This component allows C library functions, such as `fopen` and `fprintf`, to work with FS drivers. At high level, each FS driver is associated with some path prefix. When one of C library functions needs to open a file, VFS component searches for the FS driver associated with the file's 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 `/fat` prefix, and call `fopen("/fat/file.txt", "w")`. VFS component will then call `open` function of FAT driver and pass `/file.txt` argument to it (and 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, application needs to define an instance of `esp_vfs_t` structure and populate it with function pointers to FS APIs:

```
esp_vfs_t myfs = {
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
    .open = &myfs_open,
    .fstat = &myfs_fstat,
    .close = &myfs_close,
    .read = &myfs_read,
};

ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Depending on the way FS driver declares its APIs, either `read`, `write`, etc., or `read_p`, `write_p`, etc. should be used.

Case 1: API functions are declared without an extra context pointer (FS driver is a singleton):

```
ssize_t myfs_write(int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
    .flags = ESP_VFS_FLAG_DEFAULT,
    .write = &myfs_write,
```

(continues on next page)

(continued from previous page)

```
// ... other members initialized

// When registering FS, context pointer (third argument) is NULL:
ESP_ERROR_CHECK(esp_vfs_register("/data", &myfs, NULL));
```

Case 2: API functions are declared with an extra context pointer (FS driver supports multiple instances):

```
ssize_t myfs_write(myfs_t* fs, int fd, const void * data, size_t size);

// In definition of esp_vfs_t:
    .flags = ESP_VFS_FLAG_CONTEXT_PTR,
    .write_p = &myfs_write,
// ... other members initialized

// When registering FS, pass the FS context pointer into the third argument
// (hypothetical myfs_mount function is used for illustrative purposes)
myfs_t* myfs_inst1 = myfs_mount(partition1->offset, partition1->size);
ESP_ERROR_CHECK(esp_vfs_register("/data1", &myfs, myfs_inst1));

// Can register another instance:
myfs_t* myfs_inst2 = myfs_mount(partition2->offset, partition2->size);
ESP_ERROR_CHECK(esp_vfs_register("/data2", &myfs, myfs_inst2));
```

## Paths

Each registered FS has a path prefix associated with it. This prefix may be considered 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 " doesn't 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 may be used in cases when application needs to provide “fallback” filesystem, or 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. I.e. 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) may handle dots in file names differently.

When opening files, FS driver will only be given relative path to files. For example:

- myfs driver is registered with /data as path prefix
- and application calls `fopen("/data/config.json", ...)`
- then VFS component will call `myfs_open("/config.json", ...)`.
- myfs driver will open /config.json file

VFS doesn't impose a limit on total file path length, but it does limit FS path prefix to `ESP_VFS_PATH_MAX` characters. Individual FS drivers may have their own filename length limitations.

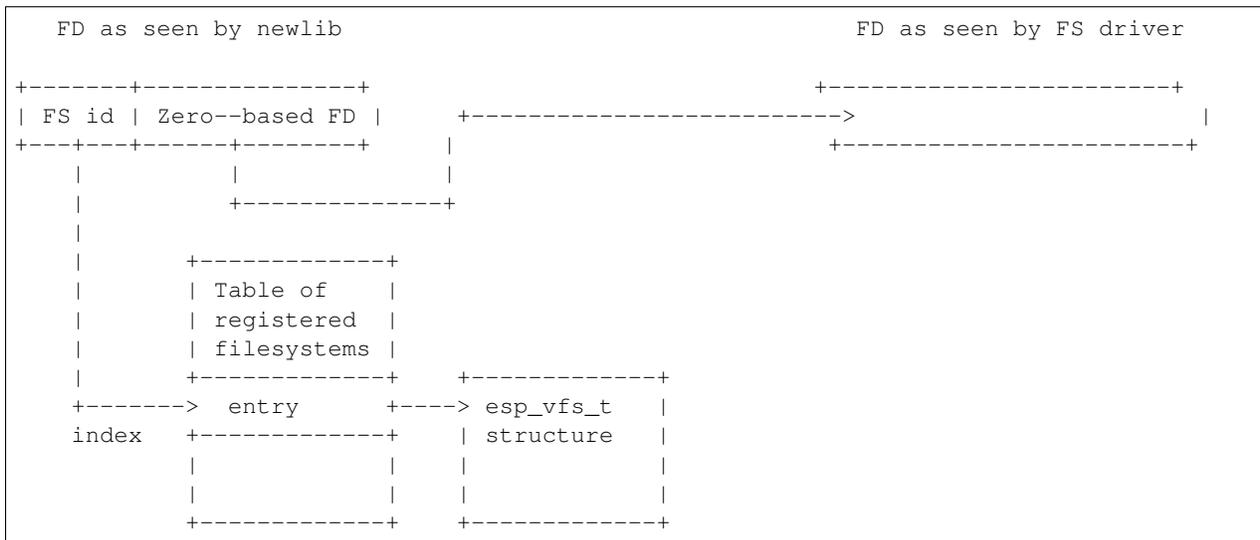
## File descriptors

It is suggested that filesystem drivers should use small positive integers as file descriptors. VFS component assumes that `CONFIG_MAX_FD_BITS` bits (12 by default) are sufficient to represent a file descriptor.

While file descriptors returned by VFS component to newlib library are rarely seen by the application, the following details may be useful for debugging purposes. File descriptors returned by VFS component are composed of two parts: FS driver ID, and the actual file descriptor. Because newlib stores file descriptors as 16-bit integers, VFS component is also limited by 16 bits to store both parts.

Lower `CONFIG_MAX_FD_BITS` bits are used to store zero-based file descriptor. The per-filesystem FD obtained from the FS `open` call, and this result is stored in the lower bits of the FD. Higher bits are used to save the index of FS in the internal table of registered filesystems.

When VFS component receives a call from newlib which has a file descriptor, this file descriptor is translated back to the FS-specific file descriptor. First, higher bits of FD are used to identify the FS. Then only the lower `CONFIG_MAX_FD_BITS` bits of the fd are masked in, and resulting FD is passed to the FS driver.



## Standard IO streams (stdin, stdout, stderr)

If “UART for console output” menuconfig option 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 and RX pin is GPIO3. These parameters can be changed in menuconfig.

Writing to `stdout` or `stderr` will send characters to the UART transmit FIFO. Reading from `stdin` will retrieve characters from the UART receive FIFO.

By default, VFS uses simple functions for reading from and writing to UART. Writes busy-wait until all data is put into UART FIFO, and reads are non-blocking, returning only the data present in the FIFO. Because of this non-blocking read behavior, higher level C library calls, such as `fscanf("%d\n", &var);` may not have desired results.

Applications which use UART driver may instruct VFS to use the driver's interrupt driven, blocking read and write functions instead. This can be done using a call to `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 optional newline conversion feature for input and output. Internally, most applications send and receive lines terminated by 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 `esp_vfs_dev_uart_set_rx_line_endings` and `esp_vfs_dev_uart_set_tx_line_endings` functions.

## 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:

```
fprintf(stderr, "42\n");
```

actually is translated to to this (by the preprocessor):

```
fprintf(__getreent()->_stderr, "42\n");
```

where the `__getreent()` function returns a per-task pointer to `struct _reent` ([newlib/include/sys/reent.h#L370-L417](#)). This structure is allocated on the TCB of each task. When a task is initialized, `_stdin`, `_stdout` and `_stderr` members of `struct _reent` are set to the values of `_stdin`, `_stdout` and `_stderr` of `_GLOBAL_REENT` (i.e. the structure which is used before FreeRTOS is started).

Such a design has the following consequences:

- It is possible to set `stdin`, `stdout`, and `stderr` for any given task without affecting other tasks, e.g. by doing `stdin = fopen("/dev/uart/1", "r")`.
- Closing default `stdin`, `stdout`, or `stderr` using `fclose` will close the FILE stream object — this will affect all other tasks.
- To change the default `stdin`, `stdout`, `stderr` streams for new tasks, modify `_GLOBAL_REENT->_stdin` (`_stdout`, `_stderr`) before creating the task.

## Application Example

Instructions

## API Reference

### Header File

- `vfs/include/esp_vfs.h`

### Functions

`ssize_t esp_vfs_write(struct _reent *r, int fd, const void *data, size_t size)`

These functions are to be used in newlib syscall table. They will be called by newlib when it needs to use any of the syscalls.

`off_t esp_vfs_lseek(struct _reent *r, int fd, off_t size, int mode)`

`ssize_t esp_vfs_read(struct _reent *r, int fd, void *dst, size_t size)`

`int esp_vfs_open(struct _reent *r, const char *path, int flags, int mode)`

int **esp\_vfs\_close** (struct \_reent \*r, int fd)

int **esp\_vfs\_fstat** (struct \_reent \*r, int fd, struct stat \*st)

int **esp\_vfs\_stat** (struct \_reent \*r, const char \*path, struct stat \*st)

int **esp\_vfs\_link** (struct \_reent \*r, const char \*n1, const char \*n2)

int **esp\_vfs\_unlink** (struct \_reent \*r, const char \*path)

int **esp\_vfs\_rename** (struct \_reent \*r, const char \*src, const char \*dst)

esp\_err\_t **esp\_vfs\_register** (const char \*base\_path, const esp\_vfs\_t \*vfs, void \*ctx)  
Register a virtual filesystem for given path prefix.

**Return** ESP\_OK if successful, ESP\_ERR\_NO\_MEM if too many VFSES are registered.

#### Parameters

- **base\_path**: file path prefix associated with the filesystem. Must be a zero-terminated C string, up to ESP\_VFS\_PATH\_MAX characters long, and at least 2 characters long. Name must start with a “/” and must not end with “/”. For example, “/data” or “/dev/spi” are valid. These VFSES would then be called to handle file paths such as “/data/myfile.txt” or “/dev/spi/0”.
- **vfs**: Pointer to *esp\_vfs\_t*, a structure which maps syscalls to the filesystem driver functions. VFS component doesn’t assume ownership of this pointer.
- **ctx**: If *vfs->flags* has ESP\_VFS\_FLAG\_CONTEXT\_PTR set, a pointer which should be passed to VFS functions. Otherwise, NULL.

esp\_err\_t **esp\_vfs\_register\_socket\_space** (const esp\_vfs\_t \*vfs, void \*ctx, int \*p\_min\_fd, int \*p\_max\_fd)

Special case function for registering a VFS that uses a method other than open() to open new file descriptors.

This is a special-purpose function intended for registering LWIP sockets to VFS.

**Return** ESP\_OK if successful, ESP\_ERR\_NO\_MEM if too many VFSES are registered.

#### 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()*.
- **p\_min\_fd**: If non-NULL, on success this variable is written with the minimum (global/user-facing) FD that this VFS will use. This is useful when ESP\_VFS\_FLAG\_SHARED\_FD\_SPACE is set in *vfs->flags*.
- **p\_max\_fd**: If non-NULL, on success this variable is written with one higher than the maximum (global/user-facing) FD that this VFS will use. This is useful when ESP\_VFS\_FLAG\_SHARED\_FD\_SPACE is set in *vfs->flags*.

esp\_err\_t **esp\_vfs\_unregister** (const char \*base\_path)  
Unregister a virtual filesystem for given path prefix

**Return** ESP\_OK if successful, ESP\_ERR\_INVALID\_STATE if VFS for given prefix hasn’t been registered

#### Parameters

- **base\_path**: file prefix previously used in *esp\_vfs\_register* call

## Structures

### **struct esp\_vfs\_t**

VFS definition structure.

This structure should be filled with pointers to corresponding FS driver functions.

VFS component will translate all FDs so that the filesystem implementation sees them starting at zero. The caller sees a global FD which is prefixed with an pre-filesystem-implementation.

Some FS implementations expect some state (e.g. pointer to some structure) to be passed in as a first argument. For these implementations, populate the members of this structure which have `_p` suffix, set flags member to `ESP_VFS_FLAG_CONTEXT_PTR` and provide the context pointer to `esp_vfs_register` function. If the implementation doesn't use this extra argument, populate the members without `_p` suffix and set flags member to `ESP_VFS_FLAG_DEFAULT`.

If the FS driver doesn't provide some of the functions, set corresponding members to `NULL`.

### Public Members

int **flags**

`ESP_VFS_FLAG_CONTEXT_PTR` or `ESP_VFS_FLAG_DEFAULT`, plus optionally `ESP_VFS_FLAG_SHARED_FD_SPACE`

## Macros

### **ESP\_VFS\_PATH\_MAX**

Maximum length of path prefix (not including zero terminator)

### **ESP\_VFS\_FLAG\_DEFAULT**

Default value of flags member in `esp_vfs_t` structure.

### **ESP\_VFS\_FLAG\_CONTEXT\_PTR**

Flag which indicates that FS needs extra context pointer in syscalls.

### **ESP\_VFS\_FLAG\_SHARED\_FD\_SPACE**

Flag which indicates that the FD space of the VFS implementation should be made the same as the FD space in newlib. This means that the normal masking off of VFS-independent fd bits is ignored and the full user-facing fd is passed to the VFS implementation.

Set the `p_minimum_fd` & `p_maximum_fd` pointers when registering the socket in order to know what range of FDs can be used with the registered VFS.

This is mostly useful for LWIP which shares the socket FD space with socket-specific functions.

## Header File

- `vfs/include/esp_vfs_dev.h`

## Functions

void **esp\_vfs\_dev\_uart\_register** ()  
add /dev/uart virtual filesystem driver

This function is called from startup code to enable serial output

void **esp\_vfs\_dev\_uart\_set\_rx\_line\_endings** (*esp\_line\_endings\_t mode*)  
Set the line endings expected to be received on UART.

This specifies the conversion between line endings received on UART and newlines (‘ ’, LF) passed into stdin:

- ESP\_LINE\_ENDINGS\_CRLF: convert CRLF to LF
- ESP\_LINE\_ENDINGS\_CR: convert CR to LF
- ESP\_LINE\_ENDINGS\_LF: no modification

**Note** this function is not thread safe w.r.t. reading from UART

#### Parameters

- mode: line endings expected on UART

void **esp\_vfs\_dev\_uart\_set\_tx\_line\_endings** (*esp\_line\_endings\_t mode*)  
Set the line endings to sent to UART.

This specifies the conversion between newlines (‘ ’, LF) on stdout and line endings sent over UART:

- ESP\_LINE\_ENDINGS\_CRLF: convert LF to CRLF
- ESP\_LINE\_ENDINGS\_CR: convert LF to CR
- ESP\_LINE\_ENDINGS\_LF: no modification

**Note** this function is not thread safe w.r.t. writing to UART

#### Parameters

- mode: line endings to send to UART

void **esp\_vfs\_dev\_uart\_use\_nonblocking** (int *uart\_num*)  
set VFS to use simple functions for reading and writing UART Read is non-blocking, write is busy waiting until TX FIFO has enough space. These functions are used by default.

#### Parameters

- uart\_num: UART peripheral number

void **esp\_vfs\_dev\_uart\_use\_driver** (int *uart\_num*)  
set VFS to use UART driver for reading and writing

**Note** application must configure UART driver before calling these functions With these functions, read and write are blocking and interrupt-driven.

#### Parameters

- uart\_num: UART peripheral number

## Enumerations

enum **esp\_line\_endings\_t**  
Line ending settings.

*Values:*

**ESP\_LINE\_ENDINGS\_CRLF**

CR + LF.

**ESP\_LINE\_ENDINGS\_CR**

CR.

**ESP\_LINE\_ENDINGS\_LF**

LF.

## 2.6.4 FAT Filesystem Support

ESP-IDF uses `FatFs` library to work with FAT filesystems. `FatFs` library resides in `fatfs` component. Although it can be used directly, many of its features can be accessed via VFS using C standard library and POSIX APIs.

Additionally, `FatFs` has been modified to support run-time pluggable disk IO layer. This allows mapping of `FatFs` drives to physical disks at run-time.

### Using FatFs with VFS

`esp_vfs_fat.h` header file defines functions to connect `FatFs` with VFS. `esp_vfs_fat_register` function 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. `esp_vfs_fat_unregister_path` function deletes the registration with VFS, and frees the `FATFS` structure.

Most applications will use the following flow when working with `esp_vfs_fat_` functions:

1. Call `esp_vfs_fat_register`, specifying path prefix where the filesystem has to be mounted (e.g. `"/sdcard"`, `"/spiflash"`), `FatFs` drive number, and a variable which will receive a pointer to `FATFS` structure.
2. Call `ff_diskio_register` function to register disk IO driver for the drive number used in step 1.
3. Call `f_mount` function (and optionally `f_fdisk`, `f_mkfs`) to mount the filesystem using the same drive number which was passed to `esp_vfs_fat_register`. See `FatFs` documentation for more details.
4. Call POSIX and C standard library functions to open, read, write, erase, copy files, etc. Use paths starting with the prefix passed to `esp_vfs_fat_register` (such as `"/sdcard/hello.txt"`).
5. Optionally, call `FatFs` library functions directly. Use paths without a VFS prefix in this case (`"/hello.txt"`).
6. Close all open files.
7. Call `f_mount` function for the same drive number, with `NULL` `FATFS*` argument, to unmount the filesystem.
8. Call `ff_diskio_register` with `NULL` `ff_diskio_impl_t*` argument and the same drive number.
9. Call `esp_vfs_fat_unregister_path` with the path where the file system is mounted to remove `FatFs` from VFS, and free the `FATFS` structure allocated on step 1.

Convenience functions, `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_sdmmc_unmount`, which wrap these steps and also handle SD card initialization, are described in the next section.

```
esp_err_t esp_vfs_fat_register(const char *base_path, const char *fat_drive, size_t max_files,
                             FATFS **out_fs)
```

Register `FATFS` with VFS component.

This function registers given FAT drive in VFS, at the specified base path. If only one drive is used, `fat_drive` argument can be an empty string. Refer to `FATFS` library documentation on how to specify FAT drive. This function also allocates `FATFS` structure which should be used for `f_mount` call.

**Note** This function doesn't mount the drive into FATFS, it just connects POSIX and C standard library IO function with FATFS. You need to mount desired drive into FATFS separately.

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if esp\_vfs\_fat\_register was already called
- ESP\_ERR\_NO\_MEM if not enough memory or too many VFSeS already registered

#### Parameters

- `base_path`: path prefix where FATFS should be registered
- `fat_drive`: FATFS drive specification; if only one drive is used, can be an empty string
- `max_files`: maximum number of files which can be open at the same time
- `out_fs`: pointer to FATFS structure which can be used for FATFS `f_mount` call is returned via this argument.

`esp_err_t esp_vfs_fat_unregister_path (const char *base_path)`

Un-register FATFS from VFS.

**Note** FATFS structure returned by `esp_vfs_fat_register` is destroyed after this call. Make sure to call `f_mount` function to unmount it before calling `esp_vfs_fat_unregister_ctx`. Difference between this function and the one above is that this one will release the correct drive, while the one above will release the last registered one

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if FATFS is not registered in VFS

#### Parameters

- `base_path`: path prefix where FATFS is registered. This is the same used when `esp_vfs_fat_register` was called

## Using FatFs with VFS and SD cards

`esp_vfs_fat.h` header file also provides a convenience function to perform steps 1–3 and 7–9, and also handle SD card initialization: `esp_vfs_fat_sdmmc_mount`. This function does only limited error handling. Developers are encouraged to look at its source code and incorporate more advanced versions into production applications. `esp_vfs_fat_sdmmc_unmount` function unmounts the filesystem and releases resources acquired by `esp_vfs_fat_sdmmc_mount`.

`esp_err_t esp_vfs_fat_sdmmc_mount (const char *base_path, const sdmmc_host_t *host_config, const void *slot_config, const esp_vfs_fat_mount_config_t *mount_config, sdmmc_card_t **out_card)`

Convenience function to get FAT filesystem on SD card registered in VFS.

This is an all-in-one function which does the following:

- initializes SDMMC driver or SPI driver with configuration in `host_config`
- initializes SD card with configuration in `slot_config`
- mounts FAT partition on SD card using FATFS library, with configuration in `mount_config`
- registers FATFS library with VFS, with prefix given by `base_prefix` variable

This function is intended to make example code more compact. For real world applications, developers should implement the logic of probing SD card, locating and mounting partition, and registering FATFS in VFS, with proper error checking and handling of exceptional conditions.

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if esp\_vfs\_fat\_sdmmc\_mount was already called
- ESP\_ERR\_NO\_MEM if memory can not be allocated
- ESP\_FAIL if partition can not be mounted
- other error codes from SDMMC or SPI drivers, SDMMC protocol, or FATFS drivers

#### Parameters

- `base_path`: path where partition should be registered (e.g. “/sdcard”)
- `host_config`: Pointer to structure describing SDMMC host. When using SDMMC peripheral, this structure can be initialized using `SDMMC_HOST_DEFAULT()` macro. When using SPI peripheral, this structure can be initialized using `SDSPI_HOST_DEFAULT()` macro.
- `slot_config`: Pointer to structure with slot configuration. For SDMMC peripheral, pass a pointer to `sdmmc_slot_config_t` structure initialized using `SDMMC_SLOT_CONFIG_DEFAULT`. For SPI peripheral, pass a pointer to `sdspi_slot_config_t` structure initialized using `SDSPI_SLOT_CONFIG_DEFAULT`.
- `mount_config`: pointer to structure with extra parameters for mounting FATFS
- `out_card`: if not NULL, pointer to the card information structure will be returned via this argument

#### **struct esp\_vfs\_fat\_mount\_config\_t**

Configuration arguments for `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_spiflash_mount` functions.

#### Public Members

##### bool `format_if_mount_failed`

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

##### int `max_files`

Max number of open files.

##### esp\_err\_t `esp_vfs_fat_sdmmc_unmount` ()

Unmount FAT filesystem and release resources acquired using `esp_vfs_fat_sdmmc_mount`.

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if `esp_vfs_fat_sdmmc_mount` hasn't been called

#### FatFS disk IO layer

FatFs has been extended with an API to register disk IO driver at runtime.

Implementation of disk IO functions for SD/MMC cards is provided. It can be registered for the given FatFs drive number using `ff_diskio_register_sdmmc` function.

void **ff\_diskio\_register** (BYTE *pdrv*, const *ff\_diskio\_impl\_t* \**discio\_impl*)

Register or unregister diskio driver for given drive number.

When FATFS library calls one of disk\_xxx functions for driver number *pdrv*, corresponding function in *discio\_impl* for given *pdrv* will be called.

#### Parameters

- *pdrv*: drive number
- *discio\_impl*: pointer to *ff\_diskio\_impl\_t* structure with diskio functions or NULL to unregister and free previously registered drive

**struct ff\_diskio\_impl\_t**

Structure of pointers to disk IO driver functions.

See FatFs documentation for details about these functions

#### Public Members

DSTATUS (\***init**) (BYTE *pdrv*)

disk initialization function

DSTATUS (\***status**) (BYTE *pdrv*)

disk status check function

DRESULT (\***read**) (BYTE *pdrv*, BYTE \**buff*, DWORD *sector*, UINT *count*)

sector read function

DRESULT (\***write**) (BYTE *pdrv*, const BYTE \**buff*, DWORD *sector*, UINT *count*)

sector write function

DRESULT (\***ioctl**) (BYTE *pdrv*, BYTE *cmd*, void \**buff*)

function to get info about disk and do some misc operations

void **ff\_diskio\_register\_sdmmc** (BYTE *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*.

## 2.6.5 Wear Levelling APIs

### Overview

Most of the flash devices and specially SPI flash devices that are used in ESP32 have sector based organization and have limited amount of erase/modification cycles per memory sector. To avoid situation when one sector reach the limit of erases when other sectors was used not often, we have made a component that avoid this situation. The wear levelling component share the amount of erases between all sectors in the memory without user interaction. The wear\_levelling component contains APIs related to reading, writing, erasing, memory mapping data in the external SPI flash through the partition component. It also has higher-level APIs which work with FAT filesystem defined in the *FAT filesystem*.

The wear levelling component, together with FAT FS component, works with FAT FS sector size 4096 bytes which is standard size of the flash devices. In this mode the component has best performance, but needs additional memory in the RAM. To save internal memory the component has two additional modes to work with sector size 512 bytes: Performance and Safety modes. In Performance mode by erase sector operation data will be stored to the RAM, sector will be erased and then data will be stored back to the flash. If by this operation power off situation will occur, the complete 4096 bytes will be lost. To prevent this the Safety mode was implemented. In safety mode the data will be first stored to the flash and after sector will be erased, will be stored back. If power off situation will occur, after power on, the data will be recovered. By default defined the sector size 512 bytes and Performance mode. To change these values please use the configuration menu.

The wear levelling component does not cache data in RAM. Write and erase functions modify flash directly, and flash contents is consistent when the function returns.

### Wear Levelling access APIs

This is the set of APIs for working with data in flash:

- `wl_mount` mount wear levelling module for defined partition
- `wl_unmount` used to unmount levelling module
- `wl_erase_range` used to erase range of addresses in flash
- `wl_write` used to write data to the partition
- `wl_read` used to read data from the partition
- `wl_size` return size of available memory in bytes
- `wl_sector_size` returns size of one sector

Generally, try to avoid using the raw wear levelling functions in favor of filesystem-specific functions.

### Memory Size

The memory size calculated in the wear Levelling module based on parameters of partition. The module use few sectors of flash for internal data.

### See also

- [FAT Filesystem](#)
- [Partition Table documentation](#)

### Application Example

An example which combines wear levelling driver with FATFS library is provided in `examples/storage/wear_levelling` directory. This example initializes the wear levelling driver, mounts FATFS partition, and writes and reads data from it using POSIX and C library APIs. See `README.md` file in the example directory for more information.

## High level API Reference

### Header Files

- `fatfs/src/esp_vfs_fat.h`

### Functions

```
esp_err_t esp_vfs_fat_spiflash_mount(const char *base_path, const char *partition_label,
                                     const esp_vfs_fat_mount_config_t *mount_config,
                                     wl_handle_t *wl_handle)
```

Convenience function to initialize FAT filesystem in SPI flash and register it in VFS.

This is an all-in-one function which does the following:

- finds the partition with defined `partition_label`. Partition label should be configured in the partition table.
- initializes flash wear levelling library on top of the given partition
- mounts FAT partition using FATFS library on top of flash wear levelling library
- registers FATFS library with VFS, with prefix given by `base_prefix` variable

This function is intended to make example code more compact.

#### Return

- `ESP_OK` on success
- `ESP_ERR_NOT_FOUND` if the partition table does not contain FATFS partition with given label
- `ESP_ERR_INVALID_STATE` if `esp_vfs_fat_spiflash_mount` was already called
- `ESP_ERR_NO_MEM` if memory can not be allocated
- `ESP_FAIL` if partition can not be mounted
- other error codes from wear levelling library, SPI flash driver, or FATFS drivers

#### Parameters

- `base_path`: path where FATFS partition should be mounted (e.g. `"/spiflash"`)
- `partition_label`: label of the partition which should be used
- `mount_config`: pointer to structure with extra parameters for mounting FATFS
- `wl_handle`: wear levelling driver handle

```
struct esp_vfs_fat_mount_config_t
```

Configuration arguments for `esp_vfs_fat_sdmmc_mount` and `esp_vfs_fat_spiflash_mount` functions.

#### Public Members

```
bool format_if_mount_failed
```

If FAT partition can not be mounted, and this parameter is true, create partition table and format the filesystem.

```
int max_files
```

Max number of open files.

`esp_err_t esp_vfs_fat_spiflash_unmount (const char *base_path, wl_handle_t wl_handle)`  
Unmount FAT filesystem and release resources acquired using `esp_vfs_fat_spiflash_mount`.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if `esp_vfs_fat_spiflash_mount` hasn't been called

**Parameters**

- `base_path`: path where partition should be registered (e.g. "/spiflash")
- `wl_handle`: wear levelling driver handle returned by `esp_vfs_fat_spiflash_mount`

## Mid level API Reference

### Header File

- `wear_levelling/include/wear_levelling.h`

### Functions

`esp_err_t wl_mount (const esp_partition_t *partition, wl_handle_t *out_handle)`  
Mount WL for defined partition.

**Return**

- ESP\_OK, if the allocation was successfully;
- ESP\_ERR\_INVALID\_ARG, if WL allocation was unsuccessful;
- ESP\_ERR\_NO\_MEM, if there was no memory to allocate WL components;

**Parameters**

- `partition`: that will be used for access
- `out_handle`: handle of the WL instance

`esp_err_t wl_unmount (wl_handle_t handle)`  
Unmount WL for defined partition.

**Return**

- ESP\_OK, if the operation completed successfully;
- or one of error codes from lower-level flash driver.

**Parameters**

- `handle`: WL partition handle

`esp_err_t wl_erase_range (wl_handle_t handle, size_t start_addr, size_t size)`  
Erase part of the WL storage.

**Return**

- ESP\_OK, if the range was erased successfully;

- ESP\_ERR\_INVALID\_ARG, if iterator or dst are NULL;
- ESP\_ERR\_INVALID\_SIZE, if erase would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

#### Parameters

- `handle`: WL handle that are related to the partition
- `start_addr`: Address where erase operation should start. Must be aligned to the result of function `wl_sector_size(...)`.
- `size`: Size of the range which should be erased, in bytes. Must be divisible by result of function `wl_sector_size(...)`.

`esp_err_t wl_write(wl_handle_t handle, size_t dest_addr, const void *src, size_t size)`

Write data to the WL storage.

Before writing data to flash, corresponding region of flash needs to be erased. This can be done using `wl_erase_range` function.

**Note** Prior to writing to WL storage, make sure it has been erased with `wl_erase_range` call.

#### Return

- ESP\_OK, if data was written successfully;
- ESP\_ERR\_INVALID\_ARG, if `dst_offset` exceeds partition size;
- ESP\_ERR\_INVALID\_SIZE, if write would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

#### Parameters

- `handle`: WL handle that are related to the partition
- `dest_addr`: Address where the data should be written, relative to the beginning of the partition.
- `src`: Pointer to the source buffer. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `size`: Size of data to be written, in bytes.

`esp_err_t wl_read(wl_handle_t handle, size_t src_addr, void *dest, size_t size)`

Read data from the WL storage.

#### Return

- ESP\_OK, if data was read successfully;
- ESP\_ERR\_INVALID\_ARG, if `src_offset` exceeds partition size;
- ESP\_ERR\_INVALID\_SIZE, if read would go out of bounds of the partition;
- or one of error codes from lower-level flash driver.

#### Parameters

- `handle`: WL module instance that was initialized before
- `dest`: Pointer to the buffer where data should be stored. Pointer must be non-NULL and buffer must be at least 'size' bytes long.
- `src_addr`: Address of the data to be read, relative to the beginning of the partition.

- `size`: Size of data to be read, in bytes.

`size_t wl_size (wl_handle_t handle)`

Get size of the WL storage.

**Return** usable size, in bytes

**Parameters**

- `handle`: WL module handle that was initialized before

`size_t wl_sector_size (wl_handle_t handle)`

Get sector size of the WL instance.

**Return** sector size, in bytes

**Parameters**

- `handle`: WL module handle that was initialized before

## Macros

`WL_INVALID_HANDLE`

## Type Definitions

```
typedef int32_t wl_handle_t
    wear levelling handle
```

## 2.6.6 SPIFFS Filesystem

### Overview

SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear leveling, file system consistency checks and more.

### Notes

- Presently, spiffs does not support directories. It produces a flat structure. If SPIFFS is mounted under `/spiffs` creating a file with path `/spiffs/tmp/myfile.txt` will create a file called `/tmp/myfile.txt` in SPIFFS, instead of `myfile.txt` under directory `/spiffs/tmp`.
- It is not a realtime stack. One write operation might last much longer than another.
- Presently, it does not detect or handle bad blocks.

### Tools

Host-Side tools for creating SPIFS partition images exist and one such tool is `mkspiffs`. You can use it to create image from a given folder and then flash that image with `esptool.py`

To do that you need to obtain some 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 partition table)
- Partition Offset: Starting address of the partition (can be obtained from partition table)

To pack a folder into 1 Megabyte image:

```
mkspiiffs -c [src_folder] -b 4096 -p 256 -s 0x100000 spiiffs.bin
```

To flash the image to ESP32 at offset 0x110000:

```
python esptool.py --chip esp32 --port [port] --baud [baud] write_flash -z 0x110000_
↪spiiffs.bin
```

## See also

- [Partition Table documentation](#)

## Application Example

An example for using SPIFFS is provided in [storage/spiiffs](#) directory. This example initializes and mounts SPIFFS partition, and writes and reads data from it using POSIX and C library APIs. See README.md file in the example directory for more information.

## High level API Reference

- [spiiffs/include/esp\\_spiiffs.h](#)

## Header File

- [spiiffs/include/esp\\_spiiffs.h](#)

## Functions

`esp_err_t esp_vfs_spiiffs_register(const esp_vfs_spiiffs_conf_t *conf)`

Register and mount SPIFFS to VFS with given path prefix.

### Return

- ESP\_OK if success
- ESP\_ERR\_NO\_MEM if objects could not be allocated
- ESP\_ERR\_INVALID\_STATE if already mounted or partition is encrypted
- ESP\_ERR\_NOT\_FOUND if partition for SPIFFS was not found
- ESP\_FAIL if mount or format fails

### Parameters

- `conf`: Pointer to *esp\_vfs\_spiiffs\_conf\_t* configuration structure

`esp_err_t esp_vfs_spiffs_unregister (const char *partition_label)`  
Unregister and unmount SPIFFS from VFS

**Return**

- ESP\_OK if successful
- ESP\_ERR\_INVALID\_STATE already unregistered

**Parameters**

- `partition_label`: Optional, label of the partition to unregister. If not specified, first partition with `subtype=spiffs` is used.

`bool esp_spiffs_mounted (const char *partition_label)`  
Check if SPIFFS is mounted

**Return**

- true if mounted
- false if not mounted

**Parameters**

- `partition_label`: Optional, label of the partition to check. If not specified, first partition with `subtype=spiffs` is used.

`esp_err_t esp_spiffs_format (const char *partition_label)`  
Format the SPIFFS partition

**Return**

- ESP\_OK if successful
- ESP\_FAIL on error

**Parameters**

- `partition_label`: Optional, label of the partition to format. If not specified, first partition with `subtype=spiffs` is used.

`esp_err_t esp_spiffs_info (const char *partition_label, size_t *total_bytes, size_t *used_bytes)`  
Get information for SPIFFS

**Return**

- ESP\_OK if success
- ESP\_ERR\_INVALID\_STATE if not mounted

**Parameters**

- `partition_label`: Optional, label of the partition to get info for. If not specified, first partition with `subtype=spiffs` is used.
- `total_bytes`: Size of the file system
- `used_bytes`: Current used bytes in the file system

## Structures

### **struct esp\_vfs\_spiffs\_conf\_t**

Configuration structure for esp\_vfs\_spiffs\_register.

#### Public Members

##### **const char \*base\_path**

File path prefix associated with the filesystem.

##### **const char \*partition\_label**

Optional, label of SPIFFS partition to use. If set to NULL, first partition with subtype=spiffs will be used.

##### **size\_t max\_files**

Maximum files that could be open at the same time.

##### **bool format\_if\_mount\_failed**

If true, it will format the file system if it fails to mount.

Example code for this API section is provided in [storage](#) directory of ESP-IDF examples.

## 2.7 System API

### 2.7.1 FreeRTOS

#### Overview

This section contains documentation of FreeRTOS types, functions, and macros. It is automatically generated from FreeRTOS header files.

For more information about FreeRTOS features specific to ESP-IDF, see [ESP-IDF FreeRTOS SMP Changes](#).

#### Task API

##### Header File

- [freertos/include/freertos/task.h](#)

##### Functions

**BaseType\_t xTaskCreatePinnedToCore** (*TaskFunction\_t pvTaskCode*, **const** char \***const** *pcName*, **const** uint32\_t *usStackDepth*, void \***const** *pvParameters*, UBaseType\_t *uxPriority*, *TaskHandle\_t* \***const** *pvCreatedTask*, **const** BaseType\_t *xCoreID*)

Create a new task with a specified affinity.

This function is similar to xTaskCreate, but allows setting task affinity in SMP system.

**Return** pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

##### Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by `configMAX_TASK_NAME_LEN` - default is 16.
- `usStackDepth`: The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and `usStackDepth` is defined as 100, 200 bytes will be allocated for stack storage.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit `portPRIVILEGE_BIT` of the priority parameter. For example, to create a privileged task at priority 2 the `uxPriority` parameter should be set to `( 2 | portPRIVILEGE_BIT )`.
- `pvCreatedTask`: Used to pass back a handle by which the created task can be referenced.
- `xCoreID`: If the value is `tskNO_AFFINITY`, the created task is not pinned to any CPU, and the scheduler can run it on any core available. Other values indicate the index number of the CPU which the task should be pinned to. Specifying values larger than `(portNUM_PROCESSORS - 1)` will cause the function to fail.

**static BaseType\_t xTaskCreate** (TaskFunction\_t *pvTaskCode*, **const** char \***const** *pcName*, **const** uint32\_t *usStackDepth*, void \***const** *pvParameters*, UBaseType\_t *uxPriority*, TaskHandle\_t \***const** *pvCreatedTask*)

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using `xTaskCreate()` then both blocks of memory are automatically dynamically allocated inside the `xTaskCreate()` function. (see <http://www.freertos.org/a00111.html>). If a task is created using `xTaskCreateStatic()` then the application writer must provide the required memory. `xTaskCreateStatic()` therefore allows a task to be created without using any dynamic memory allocation.

See `xTaskCreateStatic()` for a version that does not use any dynamic memory allocation.

`xTaskCreate()` can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using `xTaskCreateRestricted()`.

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
    static uint8_t ucParameterToPass;
    TaskHandle_t xHandle = NULL;

    // Create the task, storing the handle. Note that the passed parameter_
    ↪ucParameterToPass
```

(continues on next page)

(continued from previous page)

```

// must exist for the lifetime of the task, so in this case is declared static.
↳If it was just an
// an automatic stack variable it might no longer exist, or at least have been
↳corrupted, by the time
// the new task attempts to access it.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY,
↳ &xHandle );
    configASSERT( xHandle );

// Use the handle to delete the task.
if( xHandle != NULL )
{
    vTaskDelete( xHandle );
}
}

```

**Return** pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

#### Parameters

- pvTaskCode: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- pcName: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX\_TASK\_NAME\_LEN - default is 16.
- usStackDepth: The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.
- pvParameters: Pointer that will be used as the parameter for the task being created.
- uxPriority: The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE\_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to ( 2 | portPRIVILEGE\_BIT ).
- pvCreatedTask: Used to pass back a handle by which the created task can be referenced.

*TaskHandle\_t* **xTaskCreateStaticPinnedToCore** (TaskFunction\_t pvTaskCode, **const** char \***const** pcName, **const** uint32\_t ulStackDepth, void \***const** pvParameters, UBaseType\_t uxPriority, StackType\_t \***const** pxStackBuffer, StaticTask\_t \***const** pxTaskBuffer, **const** BaseType\_t xCoreID)

Create a new task with a specified affinity.

This function is similar to xTaskCreateStatic, but allows specifying task affinity in an SMP system.

**Return** If neither pxStackBuffer or pxTaskBuffer are NULL, then the task will be created and pdPASS is returned. If either pxStackBuffer or pxTaskBuffer are NULL then the task will not be created and errCOULD\_NOT\_ALLOCATE\_REQUIRED\_MEMORY is returned.

#### Parameters

- pvTaskCode: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by `configMAX_TASK_NAME_LEN` in `FreeRTOSConfig.h`.
- `ulStackDepth`: The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 32-bits wide and `ulStackDepth` is defined as 100 then 400 bytes will be allocated for stack storage.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task will run.
- `pxStackBuffer`: Must point to a `StackType_t` array that has at least `ulStackDepth` indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically.
- `pxTaskBuffer`: Must point to a variable of type `StaticTask_t`, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically.
- `xCoreID`: If the value is `tskNO_AFFINITY`, the created task is not pinned to any CPU, and the scheduler can run it on any core available. Other values indicate the index number of the CPU which the task should be pinned to. Specifying values larger than  $(\text{portNUM\_PROCESSORS} - 1)$  will cause the function to fail.

```
static TaskHandle_t xTaskCreateStatic (TaskFunction_t pvTaskCode, const char *const pcName,
                                        const uint32_t ulStackDepth, void *const pvParameters,
                                        UBaseType_t uxPriority, StackType_t *const pxStackBuffer,
                                        StaticTask_t *const pxTaskBuffer)
```

Create a new task and add it to the list of tasks that are ready to run.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using `xTaskCreate()` then both blocks of memory are automatically dynamically allocated inside the `xTaskCreate()` function. (see <http://www.freertos.org/a00111.html>). If a task is created using `xTaskCreateStatic()` then the application writer must provide the required memory. `xTaskCreateStatic()` therefore allows a task to be created without using any dynamic memory allocation.

Example usage:

```
// Dimensions the buffer that the task being created will use as its stack.
// NOTE: This is the number of 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( ;; )
```

(continues on next page)

(continued from previous page)

```

    {
        // 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.
        xStack,            // Array to use as the task's stack.
        &xTaskBuffer );    // Variable to hold the task's data structure.

    // puxStackBuffer and pxTaskBuffer were not NULL, so the task will have
    // been created, and xHandle will be the task's handle. Use the handle
    // to suspend the task.
    vTaskSuspend( xHandle );
}

```

**Return** If neither `pxStackBuffer` or `pxTaskBuffer` are NULL, then the task will be created and `pdPASS` is returned. If either `pxStackBuffer` or `pxTaskBuffer` are NULL then the task will not be created and `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY` is returned.

#### Parameters

- `pvTaskCode`: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- `pcName`: A descriptive name for the task. This is mainly used to facilitate debugging. The maximum length of the string is defined by `configMAX_TASK_NAME_LEN` in `FreeRTOSConfig.h`.
- `ulStackDepth`: The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 32-bits wide and `ulStackDepth` is defined as 100 then 400 bytes will be allocated for stack storage.
- `pvParameters`: Pointer that will be used as the parameter for the task being created.
- `uxPriority`: The priority at which the task will run.
- `pxStackBuffer`: Must point to a `StackType_t` array that has at least `ulStackDepth` indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically.
- `pxTaskBuffer`: Must point to a variable of type `StaticTask_t`, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically.

void **vTaskDelete** (*TaskHandle\_t xTaskToDelete*)

Remove a task from the RTOS real time kernel's management.

The task being deleted will be removed from all ready, blocked, suspended and event lists.

`INCLUDE_vTaskDelete` must be defined as 1 for this function to be available. See the configuration section for more information.

See the demo application file `death.c` for sample code that utilises `vTaskDelete()`.

**Note** The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to `vTaskDelete()`. Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

Example usage:

```
void vOtherFunction( void )
{
    TaskHandle_t xHandle;

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle_
→);

    // Use the handle to delete the task.
    vTaskDelete( xHandle );
}
```

### Parameters

- `xTaskToDelete`: The handle of the task to be deleted. Passing `NULL` will cause the calling task to be deleted.

void **vTaskDelay**( const TickType\_t *xTicksToDelay* )

Delay a task for a given number of ticks.

The actual time that the task remains blocked depends on the tick rate. The constant `portTICK_PERIOD_MS` can be used to calculate real time from the tick rate - with the resolution of one tick period.

`INCLUDE_vTaskDelay` must be defined as 1 for this function to be available. See the configuration section for more information.

`vTaskDelay()` specifies a time at which the task wishes to unblock relative to the time at which `vTaskDelay()` is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after `vTaskDelay()` is called. `vTaskDelay()` does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which `vTaskDelay()` gets called and therefore the time at which the task next executes. See `vTaskDelayUntil()` for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.

Example usage:

```
void vTaskFunction( void * pvParameters )
{
    // Block for 500ms.
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

    for( ;; )
    {
        // Simply toggle the LED every 500ms, blocking between each toggle.
        vToggleLED();
        vTaskDelay( xDelay );
    }
}
```

### Parameters

- `xTicksToDelay`: The amount of time, in tick periods, that the calling task should block.

void **vTaskDelayUntil** (TickType\_t \***const** *pxPreviousWakeTime*, **const** TickType\_t *xTimeIncrement*)  
Delay a task until a specified time.

INCLUDE\_vTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

This function can be used by periodic tasks to ensure a constant execution frequency.

This function differs from `vTaskDelay ()` in one important aspect: `vTaskDelay ()` will cause a task to block for the specified number of ticks from the time `vTaskDelay ()` is called. It is therefore difficult to use `vTaskDelay ()` by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling `vTaskDelay ()` may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas `vTaskDelay ()` specifies a wake time relative to the time at which the function is called, `vTaskDelayUntil ()` specifies the absolute (exact) time at which it wishes to unblock.

The constant `portTICK_PERIOD_MS` can be used to calculate real time from the tick rate - with the resolution of one tick period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
    TickType_t xLastWakeTime;
    const TickType_t xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount ();
    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```

### Parameters

- `pxPreviousWakeTime`: Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within `vTaskDelayUntil ()`.
- `xTimeIncrement`: The cycle time period. The task will be unblocked at time `*pxPreviousWakeTime + xTimeIncrement`. Calling `vTaskDelayUntil` with the same `xTimeIncrement` parameter value will cause the task to execute with a fixed interface period.

UBaseType\_t **uxTaskPriorityGet** (*TaskHandle\_t xTask*)  
Obtain the priority of any task.

INCLUDE\_uxTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.

Example usage:

```
void vAFunction( void )
{
    TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to obtain the priority of the created task.
    // It was created with tskIDLE_PRIORITY, but may have changed
    // it itself.
    if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
    {
        // The task has changed it's priority.
    }

    // ...

    // Is our priority higher than the created task?
    if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
    {
        // Our priority (obtained using NULL handle) is higher.
    }
}
```

**Return** The priority of xTask.

#### Parameters

- xTask: Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

UBaseType\_t **uxTaskPriorityGetFromISR** (*TaskHandle\_t* xTask)

A version of uxTaskPriorityGet() that can be used from an ISR.

**Return** The priority of xTask.

#### Parameters

- xTask: Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

*eTaskState* **eTaskGetState** (*TaskHandle\_t* xTask)

Obtain the state of any task.

States are encoded by the eTaskState enumerated type.

INCLUDE\_eTaskGetState must be defined as 1 for this function to be available. See the configuration section for more information.

**Return** The state of xTask at the time the function was called. Note the state of the task might change between the function being called, and the functions return value being tested by the calling task.

#### Parameters

- xTask: Handle of the task to be queried.

void **vTaskPrioritySet** (*TaskHandle\_t* xTask, UBaseType\_t uxNewPriority)

Set the priority of any task.

INCLUDE\_vTaskPrioritySet must be defined as 1 for this function to be available. See the configuration section for more information.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Example usage:

```
void vAFunction( void )
{
    TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to raise the priority of the created task.
    vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

    // ...

    // Use a NULL handle to raise our priority to the same value.
    vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}
```

### Parameters

- xTask: Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
- uxNewPriority: The priority to which the task will be set.

void **vTaskSuspend** (*TaskHandle\_t* xTaskToSuspend)

Suspend a task.

INCLUDE\_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

When suspended, a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Example usage:

```
void vAFunction( void )
{
    TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );
}
```

(continues on next page)

(continued from previous page)

```

// ...

// The created task will not run during this period, unless
// another task calls vTaskResume( xHandle ).

//...

// Suspend ourselves.
vTaskSuspend( NULL );

// We cannot get here unless another task calls vTaskResume
// with our handle as the parameter.
}

```

**Parameters**

- `xTaskToSuspend`: Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

void **vTaskResume** (*TaskHandle\_t* xTaskToResume)

Resumes a suspended task.

INCLUDE\_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Example usage:

```

void vAFunction( void )
{
TaskHandle_t xHandle;

// Create a task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// ...

// Use the handle to suspend the created task.
vTaskSuspend( xHandle );

// ...

// The created task will not run during this period, unless
// another task calls vTaskResume( xHandle ).

//...

// Resume the suspended task ourselves.
vTaskResume( xHandle );

// The created task will once again get microcontroller processing

```

(continues on next page)

(continued from previous page)

```

// time in accordance with its priority within the system.
}

```

**Parameters**

- xTaskToResume: Handle to the task being readied.

BaseType\_t **xTaskResumeFromISR** (*TaskHandle\_t* xTaskToResume)

An implementation of vTaskResume() that can be called from within an ISR.

INCLUDE\_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to xTaskResumeFromISR ().

xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

**Return** pdTRUE if resuming the task should result in a context switch, otherwise pdFALSE. This is used by the ISR to determine if a context switch may be required following the ISR.

**Parameters**

- xTaskToResume: Handle to the task being readied.

void **vTaskSuspendAll** (void)

Suspends the scheduler without disabling interrupts.

Context switches will not occur while the scheduler is suspended.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

API functions that have the potential to cause a context switch (for example, vTaskDelayUntil(), xQueueSend(), etc.) must not be called while the scheduler is suspended.

Example usage:

```

void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out. It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        vTaskSuspendAll ();

        // Perform the operation here. There is no need to use critical

```

(continues on next page)

(continued from previous page)

```

// sections as we have all the microcontroller processing time.
// During this time interrupts will still operate and the kernel
// tick count will be maintained.

// ...

// The operation is complete. Restart the kernel.
xTaskResumeAll ();
}
}

```

BaseType\_t **xTaskResumeAll** (void)

Resumes scheduler activity after it was suspended by a call to vTaskSuspendAll().

xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

Example usage:

```

void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out. It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        vTaskSuspendAll ();

        // Perform the operation here. There is no need to use critical
        // sections as we have all the microcontroller processing time.
        // During this time interrupts will still operate and the real
        // time kernel tick count will be maintained.

        // ...

        // The operation is complete. Restart the kernel. We want to force
        // a context switch - but there is no point if resuming the scheduler
        // caused a context switch already.
        if( !xTaskResumeAll () )
        {
            taskYIELD ();
        }
    }
}

```

**Return** If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

TickType\_t **xTaskGetTickCount** (void)

Get tick count

**Return** The count of ticks since vTaskStartScheduler was called.

TickType\_t **xTaskGetTickCountFromISR** (void)  
Get tick count from ISR

This is a version of xTaskGetTickCount() that is safe to be called from an ISR - provided that TickType\_t is the natural word size of the microcontroller being used or interrupt nesting is either not supported or not being used.

**Return** The count of ticks since vTaskStartScheduler was called.

UBaseType\_t **uxTaskGetNumberOfTasks** (void)  
Get current number of tasks

**Return** The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

char **\*pcTaskGetTaskName** (TaskHandle\_t xTaskToQuery)  
Get task name

**Return** The text (human readable) name of the task referenced by the handle xTaskToQuery. A task can query its own name by either passing in its own handle, or by setting xTaskToQuery to NULL. INCLUDE\_pcTaskGetTaskName must be set to 1 in FreeRTOSConfig.h for pcTaskGetTaskName() to be available.

UBaseType\_t **uxTaskGetStackHighWaterMark** (TaskHandle\_t xTask)  
Returns the high water mark of the stack associated with xTask.

INCLUDE\_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for this function to be available.

High water mark is the minimum free stack space there has been (in 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.

**Return** 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.

#### Parameters

- xTask: Handle of the task associated with the stack to be checked. Set xTask to NULL to check the stack of the calling task.

uint8\_t **\*pxTaskGetStackStart** (TaskHandle\_t xTask)  
Returns the start of the stack associated with xTask.

INCLUDE\_pxTaskGetStackStart must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the highest stack memory address on architectures where the stack grows down from high memory, and the lowest memory address on architectures where the stack grows up from low memory.

**Return** A pointer to the start of the stack.

#### Parameters

- xTask: Handle of the task associated with the stack returned. Set xTask to NULL to return the stack of the calling task.

void **vTaskSetApplicationTaskTag** (*TaskHandle\_t* xTask, *TaskHookFunction\_t* pxHookFunction)  
 Sets pxHookFunction to be the task hook function used by the task xTask.

**Parameters**

- xTask: Handle of the task to set the hook function for Passing xTask as NULL has the effect of setting the calling tasks hook function.
- pxHookFunction: Pointer to the hook function.

*TaskHookFunction\_t* **xTaskGetApplicationTaskTag** (*TaskHandle\_t* xTask)  
 Get the hook function assigned to given task.

**Return** The pxHookFunction value assigned to the task xTask.

**Parameters**

- xTask: Handle of the task to get the hook function for Passing xTask as NULL has the effect of getting the calling tasks hook function.

void **vTaskSetThreadLocalStoragePointer** (*TaskHandle\_t* xTaskToSet, *BaseType\_t* xIndex, void \*pvValue)

Set local storage pointer specific to the given task.

Each task contains an array of pointers that is dimensioned by the configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS setting in FreeRTOSConfig.h. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

**Parameters**

- xTaskToSet: Task to set thread local storage pointer for
- xIndex: The index of the pointer to set, from 0 to configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS - 1.
- pvValue: Pointer value to set.

void \***pvTaskGetThreadLocalStoragePointer** (*TaskHandle\_t* xTaskToQuery, *BaseType\_t* xIndex)  
 Get local storage pointer specific to the given task.

Each task contains an array of pointers that is dimensioned by the configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS setting in FreeRTOSConfig.h. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

**Return** Pointer value

**Parameters**

- xTaskToQuery: Task to get thread local storage pointer for
- xIndex: The index of the pointer to get, from 0 to configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS - 1.

void **vTaskSetThreadLocalStoragePointerAndDelCallback** (*TaskHandle\_t* xTaskToSet, *BaseType\_t* xIndex, void \*pvValue, *TlsDeleteCallbackFunction\_t* pvDelCallback)

Set local storage pointer and deletion callback.

Each task contains an array of pointers that is dimensioned by the configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS setting in FreeRTOSConfig.h. The kernel does not use the pointers itself, so the application writer can use the pointers for any purpose they wish.

Local storage pointers set for a task can reference dynamically allocated resources. This function is similar to `vTaskSetThreadLocalStoragePointer`, but provides a way to release these resources when the task gets deleted. For each pointer, a callback function can be set. This function will be called when task is deleted, with the local storage pointer index and value as arguments.

### Parameters

- `xTaskToSet`: Task to set thread local storage pointer for
- `xIndex`: The index of the pointer to set, from 0 to `configNUM_THREAD_LOCAL_STORAGE_POINTERS - 1`.
- `pvValue`: Pointer value to set.
- `pvDelCallback`: Function to call to dispose of the local storage pointer when the task is deleted.

`BaseType_t` **xTaskCallApplicationTaskHook** (*TaskHandle\_t* `xTask`, void *\*pvParameter*)

Calls the hook function associated with `xTask`. Passing `xTask` as NULL has the effect of calling the Running tasks (the calling task) hook function.

### Parameters

- `xTask`: Handle of the task to call the hook for.
- `pvParameter`: Parameter passed to the hook function for the task to interpret as it wants. The return value is the value returned by the task hook function registered by the user.

*TaskHandle\_t* **xTaskGetIdleTaskHandle** (void)

Get the handle of idle task for the current CPU.

`xTaskGetIdleTaskHandle()` is only available if `INCLUDE_xTaskGetIdleTaskHandle` is set to 1 in `FreeRTOSConfig.h`.

**Return** The handle of the idle task. It is not valid to call `xTaskGetIdleTaskHandle()` before the scheduler has been started.

*TaskHandle\_t* **xTaskGetIdleTaskHandleForCPU** (*UBaseType\_t* `cpuid`)

Get the handle of idle task for the given CPU.

`xTaskGetIdleTaskHandleForCPU()` is only available if `INCLUDE_xTaskGetIdleTaskHandle` is set to 1 in `FreeRTOSConfig.h`.

**Return** Idle task handle of a given cpu. It is not valid to call `xTaskGetIdleTaskHandleForCPU()` before the scheduler has been started.

### Parameters

- `cpuid`: The CPU to get the handle for

`UBaseType_t` **uxTaskGetSystemState** (*TaskStatus\_t* *\*const* `pxTaskStatusArray`, *const* `UBaseType_t` *uxArraySize*, *uint32\_t* *\*const* `pulTotalRunTime`)

Get the state of tasks in the system.

`configUSE_TRACE_FACILITY` must be defined as 1 in `FreeRTOSConfig.h` for `uxTaskGetSystemState()` to be available.

`uxTaskGetSystemState()` populates an `TaskStatus_t` structure for each task in the system. `TaskStatus_t` structures contain, among other things, members for the task handle, task name, task priority, task state, and total amount of run time consumed by the task. See the `TaskStatus_t` structure definition in this file for the full member list.

Example usage:

```

// This example demonstrates how a human readable table of run time stats
// information is generated from raw data provided by uxTaskGetSystemState().
// The human readable table is written to pcWriteBuffer
void vTaskGetRunTimeStats( char *pcWriteBuffer )
{
    TaskStatus_t *pxTaskStatusArray;
    volatile UBaseType_t uxArraySize, x;
    uint32_t ulTotalRunTime, ulStatsAsPercentage;

    // Make sure the write buffer does not contain a string.
    *pcWriteBuffer = 0x00;

    // Take a snapshot of the number of tasks in case it changes while this
    // function is executing.
    uxArraySize = uxTaskGetNumberOfTasks();

    // Allocate a TaskStatus_t structure for each task. An array could be
    // allocated statically at compile time.
    pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );

    if( pxTaskStatusArray != NULL )
    {
        // Generate raw status information about each task.
        uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize, &
↪ulTotalRunTime );

        // For percentage calculations.
        ulTotalRunTime /= 100UL;

        // Avoid divide by zero errors.
        if( ulTotalRunTime > 0 )
        {
            // For each populated position in the pxTaskStatusArray array,
            // format the raw data as human readable ASCII data
            for( x = 0; x < uxArraySize; x++ )
            {
                // What percentage of the total run time has the task used?
                // This will always be rounded down to the nearest integer.
                // ulTotalRunTimeDiv100 has already been divided by 100.
                ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter / ↪
↪ulTotalRunTime;

                if( ulStatsAsPercentage > 0UL )
                {
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n", ↪
↪pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter, ↪
↪ulStatsAsPercentage );
                }
                else
                {
                    // If the percentage is zero here then the task has
                    // consumed less than 1% of the total run time.
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%\r\n", ↪
↪pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter );
                }

                pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

// The array is no longer needed, free the memory it consumes.
vPortFree( pxTaskStatusArray );
}
}

```

**Note** This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

**Return** The number of TaskStatus\_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

#### Parameters

- `pxTaskStatusArray`: A pointer to an array of TaskStatus\_t structures. The array must contain at least one TaskStatus\_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
- `uxArraySize`: The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array, or the number of TaskStatus\_t structures contained in the array, not by the number of bytes in the array.
- `pulTotalRunTime`: If configGENERATE\_RUN\_TIME\_STATS is set to 1 in FreeRTOSConfig.h then \*pulTotalRunTime is set by uxTaskGetSystemState() to the total run time (as defined by the run time stats clock, see <http://www.freertos.org/rtos-run-time-stats.html>) since the target booted. pulTotalRunTime can be set to NULL to omit the total run time information.

void **vTaskList** (char \*pcWriteBuffer)

List all the current tasks.

configUSE\_TRACE\_FACILITY and configUSE\_STATS\_FORMATTING\_FUNCTIONS must both be defined as 1 for this function to be available. See the configuration section of the FreeRTOS.org website for more information.

Lists all the current tasks, along with their current state and stack usage high water mark.

**Note** This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

vTaskList() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays task names, states and stack usage.

**Note** This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskList() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskList().

### Parameters

- `pcWriteBuffer`: A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

void **vTaskGetRunTimeStats** (char \**pcWriteBuffer*)

Get the state of running tasks as a string

`configGENERATE_RUN_TIME_STATS` and `configUSE_STATS_FORMATTING_FUNCTIONS` must both be defined as 1 for this function to be available. The application must also then provide definitions for `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` and `portGET_RUN_TIME_COUNTER_VALUE()` to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

Setting `configGENERATE_RUN_TIME_STATS` to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` macro. Calling `vTaskGetRunTimeStats()` writes the total execution time of each task into a buffer, both as an absolute count value and as a percentage of the total system execution time.

**Note** This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

`vTaskGetRunTimeStats()` calls `uxTaskGetSystemState()`, then formats part of the `uxTaskGetSystemState()` output into a human readable table that displays the amount of time each task has spent in the Running state in both absolute and percentage terms.

**Note** This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

`vTaskGetRunTimeStats()` has a dependency on the `sprintf()` C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of `sprintf()` is provided in many of the FreeRTOS/Demo sub-directories in a file called `printf-stdarg.c` (note `printf-stdarg.c` does not provide a full `snprintf()` implementation!).

It is recommended that production systems call `uxTaskGetSystemState()` directly to get access to raw stats data, rather than indirectly through a call to `vTaskGetRunTimeStats()`.

### Parameters

- `pcWriteBuffer`: A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

BaseType\_t **xTaskNotify** (*TaskHandle\_t* *xTaskToNotify*, uint32\_t *ulValue*, *eNotifyAction* *eAction*)

Send task notification.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task’s notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling `xTaskNotifyWait()` or `ulTaskNotifyTake()`. If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use `xTaskNotifyWait()` to [optionally] block to wait for a notification to be pending, or `ulTaskNotifyTake()` to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

**Return** Dependent on the value of `eAction`. See the description of the `eAction` parameter.

#### Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.
- `ulValue`: Data that can be sent with the notification. How the data is used depends on the value of the `eAction` parameter.
- `eAction`: Specifies how the notification updates the task's notification value, if at all. Valid values for `eAction` are as follows:
  - `eSetBits`: The task's notification value is bitwise ORed with `ulValue`. `xTaskNotify()` always returns `pdPASS` in this case.
  - `eIncrement`: The task's notification value is incremented. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.
  - `eSetValueWithOverwrite`: The task's notification value is set to the value of `ulValue`, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). `xTaskNotify()` always returns `pdPASS` in this case.
  - `eSetValueWithoutOverwrite`: If the task being notified did not already have a notification pending then the task's notification value is set to `ulValue` and `xTaskNotify()` will return `pdPASS`. If the task being notified already had a notification pending then no action is performed and `pdFAIL` is returned.
  - `eNoAction`: The task receives a notification without its notification value being updated. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.

`BaseType_t` **xTaskNotifyFromISR** (*TaskHandle\_t* `xTaskToNotify`, *uint32\_t* `ulValue`, *eNotifyAction* `eAction`, *BaseType\_t* `*pxHigherPriorityTaskWoken`)

Send task notification from an ISR.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

A version of `xTaskNotify()` that can be used from an interrupt service routine (ISR).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling `xTaskNotifyWait()` or `ulTaskNotifyTake()`. If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use `xTaskNotifyWait()` to [optionally] block to wait for a notification to be pending, or `ulTaskNotifyTake()` to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

**Return** Dependent on the value of `eAction`. See the description of the `eAction` parameter.

### Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.
- `ulValue`: Data that can be sent with the notification. How the data is used depends on the value of the `eAction` parameter.
- `eAction`: Specifies how the notification updates the task's notification value, if at all. Valid values for `eAction` are as follows:
  - `eSetBits`: The task's notification value is bitwise ORed with `ulValue`. `xTaskNotify()` always returns `pdPASS` in this case.
  - `eIncrement`: The task's notification value is incremented. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.
  - `eSetValueWithOverwrite`: The task's notification value is set to the value of `ulValue`, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). `xTaskNotify()` always returns `pdPASS` in this case.
  - `eSetValueWithoutOverwrite`: If the task being notified did not already have a notification pending then the task's notification value is set to `ulValue` and `xTaskNotify()` will return `pdPASS`. If the task being notified already had a notification pending then no action is performed and `pdFAIL` is returned.
  - `eNoAction`: The task receives a notification without its notification value being updated. `ulValue` is not used and `xTaskNotify()` always returns `pdPASS` in this case.
- `pxHigherPriorityTaskWoken`: `xTaskNotifyFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If `xTaskNotifyFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

`BaseType_t xTaskNotifyWait` (`uint32_t ulBitsToClearOnEntry`, `uint32_t ulBitsToClearOnExit`, `uint32_t *pulNotificationValue`, `TickType_t xTicksToWait`)

Wait for task notification

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling `xTaskNotifyWait()` or `ulTaskNotifyTake()`. If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use `xTaskNotifyWait()` to [optionally] block to wait for a notification to be pending, or `ulTaskNotifyTake()` to [optionally] block to wait for its notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

**Return** If a notification was received (including notifications that were already pending when `xTaskNotifyWait` was called) then `pdPASS` is returned. Otherwise `pdFAIL` is returned.

#### Parameters

- `ulBitsToClearOnEntry`: Bits that are set in `ulBitsToClearOnEntry` value will be cleared in the calling task's notification value before the task checks to see if any notifications are pending, and optionally blocks if no notifications are pending. Setting `ulBitsToClearOnEntry` to `ULONG_MAX` (if `limits.h` is included) or `0xffffffffUL` (if `limits.h` is not included) will have the effect of resetting the task's notification value to 0. Setting `ulBitsToClearOnEntry` to 0 will leave the task's notification value unchanged.
- `ulBitsToClearOnExit`: If a notification is pending or received before the calling task exits the `xTaskNotifyWait()` function then the task's notification value (see the `xTaskNotify()` API function) is passed out using the `pulNotificationValue` parameter. Then any bits that are set in `ulBitsToClearOnExit` will be cleared in the task's notification value (note `*pulNotificationValue` is set before any bits are cleared). Setting `ulBitsToClearOnExit` to `ULONG_MAX` (if `limits.h` is included) or `0xffffffffUL` (if `limits.h` is not included) will have the effect of resetting the task's notification value to 0 before the function exits. Setting `ulBitsToClearOnExit` to 0 will leave the task's notification value unchanged when the function exits (in which case the value passed out in `pulNotificationValue` will match the task's notification value).
- `pulNotificationValue`: Used to pass the task's notification value out of the function. Note the value passed out will not be effected by the clearing of any bits caused by `ulBitsToClearOnExit` being non-zero.
- `xTicksToWait`: The maximum amount of time that the task should wait in the Blocked state for a notification to be received, should a notification not already be pending when `xTaskNotifyWait()` was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro `pdMS_TO_TICSK( value_in_ms )` can be used to convert a time specified in milliseconds to a time specified in ticks.

`void vTaskNotifyGiveFromISR (TaskHandle_t xTaskToNotify, BaseType_t *pxHigherPriorityTaskWoken)`

Simplified macro for sending task notification from ISR.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this macro to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (`uint32_t`).

A version of `xTaskNotifyGive()` that can be called from an interrupt service routine (ISR).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the

need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

`vTaskNotifyGiveFromISR()` is intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given from an ISR using the `xSemaphoreGiveFromISR()` API function, the equivalent action that instead uses a task notification is `vTaskNotifyGiveFromISR()`.

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the `ulTaskNotificationTake()` API function rather than the `xTaskNotifyWait()` API function.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for more details.

### Parameters

- `xTaskToNotify`: The handle of the task being notified. The handle to a task can be returned from the `xTaskCreate()` API function used to create the task, and the handle of the currently running task can be obtained by calling `xTaskGetCurrentTaskHandle()`.
- `pxHigherPriorityTaskWoken`: `vTaskNotifyGiveFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If `vTaskNotifyGiveFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use.

`uint32_t ulTaskNotifyTake` (`BaseType_t xClearCountOnExit`, `TickType_t xTicksToWait`)

Simplified macro for receiving task notification.

`configUSE_TASK_NOTIFICATIONS` must be undefined or defined as 1 for this function to be available.

When `configUSE_TASK_NOTIFICATIONS` is set to one each task has its own private "notification value", which is a 32-bit unsigned integer (`uint32_t`).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task's notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

`ulTaskNotifyTake()` is intended for use when a task notification is used as a faster and lighter weight binary or counting semaphore alternative. Actual FreeRTOS semaphores are taken using the `xSemaphoreTake()` API function, the equivalent action that instead uses a task notification is `ulTaskNotifyTake()`.

When a task is using its notification value as a binary or counting semaphore other tasks should send notifications to it using the `xTaskNotifyGive()` macro, or `xTaskNotify()` function with the `eAction` parameter set to `eIncrement`.

`ulTaskNotifyTake()` can either clear the task's notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task's notification value on exit, in which case the notification value acts like a counting semaphore.

A task can use `ulTaskNotifyTake()` to [optionally] block to wait for a the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

Where as `xTaskNotifyWait()` will return when a notification is pending, `ulTaskNotifyTake()` will return when the task's notification value is not zero.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for details.

**Return** The task's notification count before it is either cleared to zero or decremented (see the `xClearCountOnExit` parameter).

#### Parameters

- `xClearCountOnExit`: if `xClearCountOnExit` is `pdFALSE` then the task's notification value is decremented when the function exits. In this way the notification value acts like a counting semaphore. If `xClearCountOnExit` is not `pdFALSE` then the task's notification value is cleared to zero when the function exits. In this way the notification value acts like a binary semaphore.
- `xTicksToWait`: The maximum amount of time that the task should wait in the Blocked state for the task's notification value to be greater than zero, should the count not already be greater than zero when `ulTaskNotifyTake()` was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro `pdMS_TO_TICSK( value_in_ms )` can be used to convert a time specified in milliseconds to a time specified in ticks.

## Structures

### **struct** `xTASK_STATUS`

Used with the `uxTaskGetSystemState()` function to return the state of each task in the system.

### Public Members

#### *TaskHandle\_t* `xHandle`

The handle of the task to which the rest of the information in the structure relates.

#### **const** char \*`pcTaskName`

A pointer to the task's name. This value will be invalid if the task was deleted since the structure was populated!

#### UBaseType\_t `xTaskNumber`

A number unique to the task.

#### *eTaskState* `eCurrentState`

The state in which the task existed when the structure was populated.

#### UBaseType\_t `uxCurrentPriority`

The priority at which the task was running (may be inherited) when the structure was populated.

#### UBaseType\_t `uxBasePriority`

The priority to which the task will return if the task's current priority has been inherited to avoid unbounded priority inversion when obtaining a mutex. Only valid if `configUSE_MUTEXES` is defined as 1 in `FreeRTOSConfig.h`.

#### uint32\_t `ulRunTimeCounter`

The total run time allocated to the task so far, as defined by the run time stats clock. See <http://www.freertos.org/rtos-run-time-stats.html>. Only valid when `configGENERATE_RUN_TIME_STATS` is defined as 1 in `FreeRTOSConfig.h`.

#### StackType\_t \*`pxStackBase`

Points to the lowest address of the task's stack area.

`uint32_t usStackHighWaterMark`

The minimum amount of stack space that has remained for the task since the task was created. The closer this value is to zero the closer the task has come to overflowing its stack.

**struct xTASK\_SNAPSHOT**

Used with the `uxTaskGetSnapshotAll()` function to save memory snapshot of each task in the system. We need this struct because `TCB_t` is defined (hidden) in `tasks.c`.

## Public Members

`void *pxTCB`

Address of task control block.

`StackType_t *pxTopOfStack`

Points to the location of the last item placed on the tasks stack.

`StackType_t *pxEndOfStack`

Points to the end of the stack. `pxTopOfStack < pxEndOfStack`, stack grows hi2lo `pxTopOfStack > pxEndOfStack`, stack grows lo2hi

## Macros

`tskKERNEL_VERSION_NUMBER`

`tskKERNEL_VERSION_MAJOR`

`tskKERNEL_VERSION_MINOR`

`tskKERNEL_VERSION_BUILD`

`tskNO_AFFINITY`

`tskIDLE_PRIORITY`

Defines the priority used by the idle task. This must not be modified.

`taskYIELD()`

task. h

Macro for forcing a context switch.

`taskENTER_CRITICAL (mux)`

task. h

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

**Note** This may alter the stack (depending on the portable implementation) so must be used with care!

`taskENTER_CRITICAL_ISR (mux)`

`taskEXIT_CRITICAL (mux)`

task. h

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

**Note** This may alter the stack (depending on the portable implementation) so must be used with care!

`taskEXIT_CRITICAL_ISR (mux)`

**taskDISABLE\_INTERRUPTS** ()

task. h

Macro to disable all maskable interrupts.

**taskENABLE\_INTERRUPTS** ()

task. h

Macro to enable microcontroller interrupts.

**taskSCHEDULER\_SUSPENDED**

**taskSCHEDULER\_NOT\_STARTED**

**taskSCHEDULER\_RUNNING**

**xTaskNotifyGive** (xTaskToNotify)

Simplified macro for sending task notification.

configUSE\_TASK\_NOTIFICATIONS must be undefined or defined as 1 for this macro to be available.

When configUSE\_TASK\_NOTIFICATIONS is set to one each task has its own private “notification value”, which is a 32-bit unsigned integer (uint32\_t).

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment the task’s notification value. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

xTaskNotifyGive() is a helper macro intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given using the xSemaphoreGive() API function, the equivalent action that instead uses a task notification is xTaskNotifyGive().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTake() API function rather than the xTaskNotifyWait() API function.

See <http://www.FreeRTOS.org/RTOS-task-notifications.html> for more details.

**Return** xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement - so pdPASS is always returned.

#### Parameters

- xTaskToNotify: The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle().

## Type Definitions

**typedef** void \***TaskHandle\_t**

task. h

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an TaskHandle\_t variable that can then be used as a parameter to vTaskDelete to delete the task.

**typedef** BaseType\_t (\***TaskHookFunction\_t**) (void \*)

Defines the prototype to which the application task hook function must conform.

**typedef struct *xTASK\_STATUS* TaskStatus\_t**

Used with the uxTaskGetSystemState() function to return the state of each task in the system.

**typedef struct *xTASK\_SNAPSHOT* TaskSnapshot\_t**

Used with the uxTaskGetSnapshotAll() function to save memory snapshot of each task in the system. We need this struct because TCB\_t is defined (hidden) in tasks.c.

**typedef void (\*TlsDeleteCallbackFunction\_t) (int, void \*)**

Prototype of local storage pointer deletion callback.

## Enumerations

**enum eTaskState**

Task states returned by eTaskGetState.

*Values:*

**eRunning = 0**

A task is querying the state of itself, so must be running.

**eReady**

The task being queried is in a read or pending ready list.

**eBlocked**

The task being queried is in the Blocked state.

**eSuspended**

The task being queried is in the Suspended state, or is in the Blocked state with an infinite time out.

**eDeleted**

The task being queried has been deleted, but its TCB has not yet been freed.

**enum eNotifyAction**

Actions that can be performed when vTaskNotify() is called.

*Values:*

**eNoAction = 0**

Notify the task without updating its notify value.

**eSetBits**

Set bits in the task's notification value.

**eIncrement**

Increment the task's notification value.

**eSetValueWithOverwrite**

Set the task's notification value to a specific value even if the previous value has not yet been read by the task.

**eSetValueWithoutOverwrite**

Set the task's notification value if the previous value has been read by the task.

**enum eSleepModeStatus**

Possible return values for eTaskConfirmSleepModeStatus().

*Values:*

**eAbortSleep = 0**

A task has been made ready or a context switch pended since portSUPPORESS\_TICKS\_AND\_SLEEP() was called - abort entering a sleep mode.

**eStandardSleep**

Enter a sleep mode that will not last any longer than the expected idle time.

**eNoTasksWaitingTimeout**

No tasks are waiting for a timeout so it is safe to enter a sleep mode that can only be exited by an external interrupt.

**Queue API****Header File**

- freertos/include/freertos/queue.h

**Functions**

BaseType\_t **xQueueGenericSendFromISR**(*QueueHandle\_t* xQueue, const void \*const pvItemToQueue, BaseType\_t \*const pxHigherPriorityTaskWoken, const BaseType\_t xCopyPosition)

It is preferred that the macros xQueueSendFromISR(), xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() be used in place of calling this function directly. xQueueGiveFromISR() is an equivalent for use by semaphores that don't actually copy any data.

Post an item on a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
    char cIn;
    BaseType_t xHigherPriorityTaskWokenByPost;

    // We have not woken a task at the start of the ISR.
    xHigherPriorityTaskWokenByPost = pdFALSE;

    // Loop until the buffer is empty.
    do
    {
        // Obtain a byte from the buffer.
        cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

        // Post each byte.
        xQueueGenericSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWokenByPost,
↳queueSEND_TO_BACK );

    } while( portINPUT_BYTE( BUFFER_COUNT ) );

    // Now the buffer is empty we can switch context if necessary. Note that the
    // name of the yield function required is port specific.
    if( xHigherPriorityTaskWokenByPost )
    {
        taskYIELD_YIELD_FROM_ISR();
    }
}
```

**Return** pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE\_FULL.

#### Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- `pxHigherPriorityTaskWoken`: `xQueueGenericSendFromISR()` will set `*pxHigherPriorityTaskWoken` to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueGenericSendFromISR()` sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.
- `xCopyPosition`: Can take the value `queueSEND_TO_BACK` to place the item at the back of the queue, or `queueSEND_TO_FRONT` to place the item at the front of the queue (for high priority messages).

BaseType\_t **xQueueGiveFromISR** (*QueueHandle\_t* xQueue, BaseType\_t \*const pxHigherPriorityTaskWoken)

BaseType\_t **xQueueIsQueueEmptyFromISR** (const *QueueHandle\_t* xQueue)

Utilities to query queues that are safe to use from an ISR. These utilities should be used only from within an ISR, or within a critical section.

BaseType\_t **xQueueIsQueueFullFromISR** (const *QueueHandle\_t* xQueue)

UBaseType\_t **uxQueueMessagesWaitingFromISR** (const *QueueHandle\_t* xQueue)

BaseType\_t **xQueueGenericSend** (*QueueHandle\_t* xQueue, const void \*const pvItemToQueue, TickType\_t xTicksToWait, const BaseType\_t xCopyPosition)

It is preferred that the macros `xQueueSend()`, `xQueueSendToFront()` and `xQueueSendToBack()` are used in place of calling this function directly.

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See `xQueueSendFromISR()` for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
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 * ) );

// ...
```

(continues on next page)

(continued from previous page)

```

if( xQueue1 != 0 )
{
    // Send an uint32_t. Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10,
↪queueSEND_TO_BACK ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueGenericSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0,
↪queueSEND_TO_BACK );
}

// ... Rest of task code.
}

```

**Return** pdTRUE if the item was successfully posted, otherwise errQUEUE\_FULL.

#### Parameters

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- xTicksToWait: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK\_PERIOD\_MS should be used to convert to real time if this is required.
- xCopyPosition: Can take the value queueSEND\_TO\_BACK to place the item at the back of the queue, or queueSEND\_TO\_FRONT to place the item at the front of the queue (for high priority messages).

BaseType\_t **xQueuePeekFromISR** (*QueueHandle\_t* xQueue, void \***const** pvBuffer)

A version of xQueuePeek() that can be called from an interrupt service routine (ISR).

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueueReceive().

**Return** pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

#### Parameters

- xQueue: The handle to the queue from which the item is to be received.

- `pvBuffer`: Pointer to the buffer into which the received item will be copied.

`BaseType_t xQueueGenericReceive` (*QueueHandle\_t* `xQueue`, `void *const pvBuffer`, *TickType\_t* `xTicksToWait`, `const BaseType_t xJustPeek`)

It is preferred that the macro `xQueueReceive()` be used rather than calling this function directly.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

This function must not be used in an interrupt service routine. See `xQueueReceiveFromISR` for an alternative that can.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
    struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = &xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

    // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxdMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueGenericReceive( xQueue, &pxRxdMessage, ( TickType_t ) 10 ) )
        {
            // pxRxdMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

}

// ... Rest of task code.
}

```

**Return** pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

#### Parameters

- **xQueue**: The handle to the queue from which the item is to be received.
- **pvBuffer**: Pointer to the buffer into which the received item will be copied.
- **xTicksToWait**: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK\_PERIOD\_MS should be used to convert to real time if this is required. xQueueGenericReceive() will return immediately if the queue is empty and xTicksToWait is 0.
- **xJustPeek**: When set to true, the item received from the queue is not actually removed from the queue - meaning a subsequent call to xQueueReceive() will return the same item. When set to false, the item being received from the queue is also removed from the queue.

UBaseType\_t **uxQueueMessagesWaiting** (const *QueueHandle\_t* xQueue)

Return the number of messages stored in a queue.

**Return** The number of messages available in the queue.

#### Parameters

- **xQueue**: A handle to the queue being queried.

UBaseType\_t **uxQueueSpacesAvailable** (const *QueueHandle\_t* xQueue)

Return the number of free spaces available in a queue. This is equal to the number of items that can be sent to the queue before the queue becomes full if no items are removed.

**Return** The number of spaces available in the queue.

#### Parameters

- **xQueue**: A handle to the queue being queried.

void **vQueueDelete** (*QueueHandle\_t* xQueue)

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

#### Parameters

- **xQueue**: A handle to the queue to be deleted.

BaseType\_t **xQueueReceiveFromISR** (*QueueHandle\_t* xQueue, void \*const pvBuffer, BaseType\_t \*const pxHigherPriorityTaskWoken)

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Example usage:

```

QueueHandle_t xQueue;

// Function to create a queue and post some values.
void vAFunction( void *pvParameters )

```

(continues on next page)

(continued from previous page)

```

{
char cValueToPost;
const TickType_t xTicksToWait = ( TickType_t )0xffff;

    // Create a queue capable of containing 10 characters.
    xQueue = xQueueCreate( 10, sizeof( char ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Post some characters that will be used within an ISR.  If the queue
    // is full then this task will block for xTicksToWait ticks.
    cValueToPost = 'a';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
    cValueToPost = 'b';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );

    // ... keep posting characters ... this task may block when the queue
    // becomes full.

    cValueToPost = 'c';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
}

// ISR that outputs all the characters received on the queue.
void vISR_Routine( void )
{
    BaseType_t xTaskWokenByReceive = pdFALSE;
    char cRxdChar;

    while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxdChar, &
    ↪xTaskWokenByReceive) )
    {
        // A character was received.  Output the character now.
        vOutputCharacter( cRxdChar );

        // If removing the character from the queue woke the task that was
        // posting onto the queue cTaskWokenByReceive will have been set to
        // pdTRUE.  No matter how many times this loop iterates only one
        // task will be woken.
    }

    if( cTaskWokenByPost != ( char ) pdFALSE;
    {
        taskYIELD ();
    }
}

```

**Return** pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

#### Parameters

- xQueue: The handle to the queue from which the item is to be received.
- pvBuffer: Pointer to the buffer into which the received item will be copied.

- `pxHigherPriorityTaskWoken`: A task may be blocked waiting for space to become available on the queue. If `xQueueReceiveFromISR` causes such a task to unblock `*pxTaskWoken` will get set to `pdTRUE`, otherwise `*pxTaskWoken` will remain unchanged.

void **vQueueAddToRegistry** (*QueueHandle\_t* xQueue, const char \*pcName)

The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call `vQueueAddToRegistry()` add a queue, semaphore or mutex handle to the registry if you want the handle to be available to a kernel aware debugger. If you are not using a kernel aware debugger then this function can be ignored.

`configQUEUE_REGISTRY_SIZE` defines the maximum number of handles the registry can hold. `configQUEUE_REGISTRY_SIZE` must be greater than 0 within `FreeRTOSConfig.h` for the registry to be available. Its value does not effect the number of queues, semaphores and mutexes that can be created - just the number that the registry can hold.

#### Parameters

- `xQueue`: The handle of the queue being added to the registry. This is the handle returned by a call to `xQueueCreate()`. Semaphore and mutex handles can also be passed in here.
- `pcName`: The name to be associated with the handle. This is the name that the kernel aware debugger will display. The queue registry only stores a pointer to the string - so the string must be persistent (global or preferably in ROM/Flash), not on the stack.

void **vQueueUnregisterQueue** (*QueueHandle\_t* xQueue)

The registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call `vQueueAddToRegistry()` add a queue, semaphore or mutex handle to the registry if you want the handle to be available to a kernel aware debugger, and `vQueueUnregisterQueue()` to remove the queue, semaphore or mutex from the register. If you are not using a kernel aware debugger then this function can be ignored.

#### Parameters

- `xQueue`: The handle of the queue being removed from the registry.

const char \***pcQueueGetName** (*QueueHandle\_t* xQueue)

The queue registry is provided as a means for kernel aware debuggers to locate queues, semaphores and mutexes. Call `pcQueueGetName()` to look up and return the name of a queue in the queue registry from the queue's handle.

**Note** This function has been back ported from FreeRTOS v9.0.0

**Return** If the queue is in the registry then a pointer to the name of the queue is returned. If the queue is not in the registry then NULL is returned.

#### Parameters

- `xQueue`: The handle of the queue the name of which will be returned.

*QueueHandle\_t* **xQueueGenericCreate** (const UBaseType\_t uxQueueLength, const UBaseType\_t uxItemSize, const uint8\_t ucQueueType)

Generic version of the function used to create a queue using dynamic memory allocation. This is called by other functions and macros that create other RTOS objects that use the queue structure as their base.

*QueueHandle\_t* **xQueueGenericCreateStatic** (const UBaseType\_t uxQueueLength, const UBaseType\_t uxItemSize, uint8\_t \*pucQueueStorage, StaticQueue\_t \*pxStaticQueue, const uint8\_t ucQueueType)

Generic version of the function used to create a queue using dynamic memory allocation. This is called by other functions and macros that create other RTOS objects that use the queue structure as their base.

*QueueSetHandle\_t* **xQueueCreateSet** (*const* *UBaseType\_t* *uxEventQueueLength*)

Queue sets provide a mechanism to allow a task to block (pend) on a read operation from multiple queues or semaphores simultaneously.

See FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c for an example using this function.

A queue set must be explicitly created using a call to `xQueueCreateSet()` before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls to `xQueueAddToSet()`. `xQueueSelectFromSet()` is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.

Note 1: See the documentation on <http://www.FreeRTOS.org/RTOS-queue-sets.html> for reasons why queue sets are very rarely needed in practice as there are simpler methods of blocking on multiple objects.

Note 2: Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

Note 3: An additional 4 bytes of RAM is required for each space in a every queue added to a queue set. Therefore counting semaphores that have a high maximum count value should not be added to a queue set.

Note 4: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to `xQueueSelectFromSet()` has first returned a handle to that set member.

**Return** If the queue set is created successfully then a handle to the created queue set is returned. Otherwise NULL is returned.

#### Parameters

- *uxEventQueueLength*: Queue sets store events that occur on the queues and semaphores contained in the set. *uxEventQueueLength* specifies the maximum number of events that can be queued at once. To be absolutely certain that events are not lost *uxEventQueueLength* should be set to the total sum of the length of the queues added to the set, where binary semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. Examples:
  - If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then *uxEventQueueLength* should be set to (5 + 12 + 1), or 18.
  - If a queue set is to hold three binary semaphores then *uxEventQueueLength* should be set to (1 + 1 + 1), or 3.
  - If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then *uxEventQueueLength* should be set to (5 + 3), or 8.

*BaseType\_t* **xQueueAddToSet** (*QueueSetMemberHandle\_t* *xQueueOrSemaphore*, *QueueSetHandle\_t* *xQueueSet*)

Adds a queue or semaphore to a queue set that was previously created by a call to `xQueueCreateSet()`.

See FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c for an example using this function.

Note 1: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to `xQueueSelectFromSet()` has first returned a handle to that set member.

**Return** If the queue or semaphore was successfully added to the queue set then `pdPASS` is returned. If the queue could not be successfully added to the queue set because it is already a member of a different queue set then `pdFAIL` is returned.

#### Parameters

- *xQueueOrSemaphore*: The handle of the queue or semaphore being added to the queue set (cast to an `QueueSetMemberHandle_t` type).

- `xQueueSet`: The handle of the queue set to which the queue or semaphore is being added.

`BaseType_t xQueueRemoveFromSet (QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet)`

Removes a queue or semaphore from a queue set. A queue or semaphore can only be removed from a set if the queue or semaphore is empty.

See `FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c` for an example using this function.

**Return** If the queue or semaphore was successfully removed from the queue set then `pdPASS` is returned. If the queue was not in the queue set, or the queue (or semaphore) was not empty, then `pdFAIL` is returned.

#### Parameters

- `xQueueOrSemaphore`: The handle of the queue or semaphore being removed from the queue set (cast to an `QueueSetMemberHandle_t` type).
- `xQueueSet`: The handle of the queue set in which the queue or semaphore is included.

`QueueSetMemberHandle_t xQueueSelectFromSet (QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait)`

`xQueueSelectFromSet()` selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). `xQueueSelectFromSet()` effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.

See `FreeRTOS/Source/Demo/Common/Minimal/QueueSet.c` for an example using this function.

Note 1: See the documentation on <http://www.FreeRTOS.org/RTOS-queue-sets.html> for reasons why queue sets are very rarely needed in practice as there are simpler methods of blocking on multiple objects.

Note 2: Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

Note 3: A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to `xQueueSelectFromSet()` has first returned a handle to that set member.

**Return** `xQueueSelectFromSet()` will return the handle of a queue (cast to a `QueueSetMemberHandle_t` type) contained in the queue set that contains data, or the handle of a semaphore (cast to a `QueueSetMemberHandle_t` type) contained in the queue set that is available, or `NULL` if no such queue or semaphore exists before before the specified block time expires.

#### Parameters

- `xQueueSet`: The queue set on which the task will (potentially) block.
- `xTicksToWait`: The maximum time, in ticks, that the calling task will remain in the Blocked state (with other tasks executing) to wait for a member of the queue set to be ready for a successful queue read or semaphore take operation.

`QueueSetMemberHandle_t xQueueSelectFromSetFromISR (QueueSetHandle_t xQueueSet)`

A version of `xQueueSelectFromSet()` that can be used from an ISR.

## Macros

`xQueueCreate (uxQueueLength, uxItemSize)`

Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
};

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );
if( xQueue1 == 0 )
{
// Queue was not created and must not be used.
}

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
if( xQueue2 == 0 )
{
// Queue was not created and must not be used.
}

// ... Rest of task code.
}

```

**Return** If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

#### Parameters

- `uxQueueLength`: The maximum number of items that the queue can contain.
- `uxItemSize`: The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

#### **xQueueCreateStatic** (uxQueueLength, uxItemSize, pucQueueStorage, pxQueueBuffer)

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using `xQueueCreate()` then both blocks of memory are automatically dynamically allocated inside the `xQueueCreate()` function. (see <http://www.freertos.org/a00111.html>). If a queue is created using `xQueueCreateStatic()` then the application writer must provide the memory that will get used by the queue. `xQueueCreateStatic()` therefore allows a queue to be created without using any dynamic memory allocation.

<http://www.FreeRTOS.org/Embedded-RTOS-Queues.html>

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
};

```

(continues on next page)

(continued from previous page)

```

#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )

// xQueueBuffer will hold the queue structure.
StaticQueue_t xQueueBuffer;

// ucQueueStorage will hold the items posted to the queue. Must be at least
// [(queue length) * ( queue item size)] bytes long.
uint8_t ucQueueStorage[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )
{
    QueueHandle_t xQueue1;

    // Create a queue capable of containing 10 uint32_t values.
    xQueue1 = xQueueCreate( QUEUE_LENGTH, // The number of items the queue can hold.
                           ITEM_SIZE    // The size of each item in the queue
                           &( ucQueueStorage[ 0 ] ), // The buffer that will hold
    →the items in the queue.
                           &xQueueBuffer ); // The buffer that will hold the queue
    →structure.

    // The queue is guaranteed to be created successfully as no dynamic memory
    // allocation is used. Therefore xQueue1 is now a handle to a valid queue.

    // ... Rest of task code.
}

```

**Return** If the queue is created then a handle to the created queue is returned. If `pxQueueBuffer` is NULL then NULL is returned.

#### Parameters

- `uxQueueLength`: The maximum number of items that the queue can contain.
- `uxItemSize`: The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.
- `pucQueueStorage`: If `uxItemSize` is not zero then `pucQueueStorageBuffer` must point to a `uint8_t` array that is at least large enough to hold the maximum number of items that can be in the queue at any one time - which is  $( uxQueueLength * uxItemSize )$  bytes. If `uxItemSize` is zero then `pucQueueStorageBuffer` can be NULL.
- `pxQueueBuffer`: Must point to a variable of type `StaticQueue_t`, which will be used to hold the queue's data structure.

**xQueueSendToFront** (`xQueue`, `pvItemToQueue`, `xTicksToWait`)

This is a macro that calls `xQueueGenericSend()`.

Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See `xQueueSendFromISR ()` for an alternative which may be used in an ISR.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
    // Send an uint32_t. Wait for 10 ticks for space to become
    // available if necessary.
    if( xQueueSendToFront( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
    {
        // Failed to post the message, even after 10 ticks.
    }
}

if( xQueue2 != 0 )
{
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = &xMessage;
    xQueueSendToFront( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}

// ... Rest of task code.
}

```

**Return** `pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

#### Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- `xTicksToWait`: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required.

**xQueueSendToBack** (xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls xQueueGenericSend().

Post an item to the back of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR () for an alternative which may be used in an ISR.

Example usage:

```

struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
// Send an uint32_t. Wait for 10 ticks for space to become
// available if necessary.
if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
{
// Failed to post the message, even after 10 ticks.
}
}

if( xQueue2 != 0 )
{
// Send a pointer to a struct AMessage object. Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSendToBack( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}

// ... Rest of task code.
}

```

**Return** pdTRUE if the item was successfully posted, otherwise errQUEUE\_FULL.

#### Parameters

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items

the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

- `xTicksToWait`: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required.

#### **xQueueSend** (xQueue, pvItemToQueue, xTicksToWait)

This is a macro that calls `xQueueGenericSend()`. It is included for backward compatibility with versions of FreeRTOS.org that did not include the `xQueueSendToFront()` and `xQueueSendToBack()` macros. It is equivalent to `xQueueSendToBack()`.

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See `xQueueSendFromISR ()` for an alternative which may be used in an ISR.

Example usage:

```
struct AMessage
{
char ucMessageID;
char ucData[ 20 ];
} xMessage;

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

// Create a queue capable of containing 10 uint32_t values.
xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

// Create a queue capable of containing 10 pointers to AMessage structures.
// These should be passed by pointer as they contain a lot of data.
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

// ...

if( xQueue1 != 0 )
{
// Send an uint32_t. Wait for 10 ticks for space to become
// available if necessary.
if( xQueueSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
{
// Failed to post the message, even after 10 ticks.
}
}

if( xQueue2 != 0 )
{
// Send a pointer to a struct AMessage object. Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
}
}
```

(continues on next page)

(continued from previous page)

```
// ... Rest of task code.
}
```

**Return** pdTRUE if the item was successfully posted, otherwise errQUEUE\_FULL.

#### Parameters

- **xQueue**: The handle to the queue on which the item is to be posted.
- **pvItemToQueue**: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- **xTicksToWait**: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK\_PERIOD\_MS should be used to convert to real time if this is required.

#### **xQueueOverwrite** (xQueue, pvItemToQueue)

Only for use with queues that have a length of one - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR () for an alternative which may be used in an ISR.

Example usage:

```
void vFunction( void *pvParameters )
{
    QueueHandle_t xQueue;
    uint32_t ulVarToSend, ulValReceived;

    // Create a queue to hold one uint32_t value. It is strongly
    // recommended *not* to use xQueueOverwrite() on queues that can
    // contain more than one value, and doing so will trigger an assertion
    // if configASSERT() is defined.
    xQueue = xQueueCreate( 1, sizeof( uint32_t ) );

    // Write the value 10 to the queue using xQueueOverwrite().
    ulVarToSend = 10;
    xQueueOverwrite( xQueue, &ulVarToSend );

    // Peeking the queue should now return 10, but leave the value 10 in
    // the queue. A block time of zero is used as it is known that the
    // queue holds a value.
    ulValReceived = 0;
    xQueuePeek( xQueue, &ulValReceived, 0 );

    if( ulValReceived != 10 )
    {
        // Error unless the item was removed by a different task.
    }

    // The queue is still full. Use xQueueOverwrite() to overwrite the
    // value held in the queue with 100.
    ulVarToSend = 100;
```

(continues on next page)

(continued from previous page)

```

xQueueOverwrite( xQueue, &ulVarToSend );

// This time read from the queue, leaving the queue empty once more.
// A block time of 0 is used again.
xQueueReceive( xQueue, &ulValReceived, 0 );

// The value read should be the last value written, even though the
// queue was already full when the value was written.
if( ulValReceived != 100 )
{
    // Error!
}

// ...
}

```

**Return** `xQueueOverwrite()` is a macro that calls `xQueueGenericSend()`, and therefore has the same return values as `xQueueSendToFront()`. However, `pdPASS` is the only value that can be returned because `xQueueOverwrite()` will write to the queue even when the queue is already full.

#### Parameters

- `xQueue`: The handle of the queue to which the data is being sent.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

#### **xQueuePeek** (xQueue, pvBuffer, xTicksToWait)

This is a macro that calls the `xQueueGenericReceive()` function.

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to `xQueueReceive()`.

This macro must not be used in an interrupt service routine. See `xQueuePeekFromISR()` for an alternative that can be called from an interrupt service routine.

Example usage:

```

struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
    struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.

```

(continues on next page)

(continued from previous page)

```

xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
if( xQueue == 0 )
{
    // Failed to create the queue.
}

// ...

// Send a pointer to a struct AMessage object. Don't block if the
// queue is already full.
pxMessage = & xMessage;
xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

// ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxdMessage;

    if( xQueue != 0 )
    {
        // Peek a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueuePeek( xQueue, &( pxRxdMessage ), ( TickType_t ) 10 ) )
        {
            // pxRxdMessage now points to the struct AMessage variable posted
            // by vATask, but the item still remains on the queue.
        }
    }

    // ... Rest of task code.
}

```

**Return** pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

#### Parameters

- **xQueue**: The handle to the queue from which the item is to be received.
- **pvBuffer**: Pointer to the buffer into which the received item will be copied.
- **xTicksToWait**: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK\_PERIOD\_MS should be used to convert to real time if this is required. xQueuePeek() will return immediately if xTicksToWait is 0 and the queue is empty.

**xQueueReceive** (xQueue, pvBuffer, xTicksToWait)

queue. h

This is a macro that calls the xQueueGenericReceive() function.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items are removed from the queue.

This function must not be used in an interrupt service routine. See `xQueueReceiveFromISR` for an alternative that can.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
    struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

    // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxdMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxdMessage ), ( TickType_t ) 10 ) )
        {
            // pxRxdMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }

    // ... Rest of task code.
}
```

**Return** `pdTRUE` if an item was successfully received from the queue, otherwise `pdFALSE`.

**Parameters**

- `xQueue`: The handle to the queue from which the item is to be received.
- `pvBuffer`: Pointer to the buffer into which the received item will be copied.
- `xTicksToWait`: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. `xQueueReceive()` will return immediately if `xTicksToWait` is zero and the queue is empty. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required.

**`xQueueSendToFrontFromISR`** (`xQueue`, `pvItemToQueue`, `pxHigherPriorityTaskWoken`)

This is a macro that calls `xQueueGenericSendFromISR()`.

Post an item to the front of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
 BaseType_t xHigherPriorityTaskWoken;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWoken = pdFALSE;

// Loop until the buffer is empty.
do
{
// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post the byte.
xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
portYIELD_FROM_ISR ();
}
}
```

**Return** `pdTRUE` if the data was successfully sent to the queue, otherwise `errQUEUE_FULL`.

#### Parameters

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- `pxHigherPriorityTaskWoken`: `xQueueSendToFrontFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueSendToFromFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited.

**xQueueSendToBackFromISR** (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the back of a queue. It is safe to use this macro from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
 BaseType_t xHigherPriorityTaskWoken;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWoken = pdFALSE;

// Loop until the buffer is empty.
do
{
// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post the byte.
xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
portYIELD_FROM_ISR ();
}
}
```

**Return** pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE\_FULL.

**Parameters**

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- pxHigherPriorityTaskWoken: xQueueSendToBackFromISR() will set \*pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToBackFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

**xQueueOverwriteFromISR** (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

A version of xQueueOverwrite() that can be used in an interrupt service routine (ISR).

Only for use with queues that can hold a single item - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

Example usage:

```

QueueHandle_t xQueue;

void vFunction( void *pvParameters )
{
    // Create a queue to hold one uint32_t value. It is strongly
    // recommended *not* to use xQueueOverwriteFromISR() on queues that can
    // contain more than one value, and doing so will trigger an assertion
    // if configASSERT() is defined.
    xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}

void vAnInterruptHandler( void )
{
    // xHigherPriorityTaskWoken must be set to pdFALSE before it is used.
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    uint32_t ulVarToSend, ulValReceived;

    // Write the value 10 to the queue using xQueueOverwriteFromISR().
    ulVarToSend = 10;
    xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

    // The queue is full, but calling xQueueOverwriteFromISR() again will still
    // pass because the value held in the queue will be overwritten with the
    // new value.
    ulVarToSend = 100;
    xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

    // Reading from the queue will now return 100.

    // ...

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        // Writing to the queue caused a task to unblock and the unblocked task
        // has a priority higher than or equal to the priority of the currently
        // executing task (the task this interrupt interrupted). Perform a ↵
        ↵context
        // switch so this interrupt returns directly to the unblocked task.
        portYIELD_FROM_ISR(); // or portEND_SWITCHING_ISR() depending on the port.
    }
}

```

**Return** xQueueOverwriteFromISR() is a macro that calls xQueueGenericSendFromISR(), and therefore has the same return values as xQueueSendToFrontFromISR(). However, pdPASS is the only value that can be returned because xQueueOverwriteFromISR() will write to the queue even when the queue is already full.

#### Parameters

- xQueue: The handle to the queue on which the item is to be posted.
- pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
- pxHigherPriorityTaskWoken: xQueueOverwriteFromISR() will set \*pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueOverwriteFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

**xQueueSendFromISR** (xQueue, pvItemToQueue, pxHigherPriorityTaskWoken)

This is a macro that calls `xQueueGenericSendFromISR()`. It is included for backward compatibility with versions of FreeRTOS.org that did not include the `xQueueSendToBackFromISR()` and `xQueueSendToFrontFromISR()` macros.

Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
 BaseType_t xHigherPriorityTaskWoken;

// We have not woken a task at the start of the ISR.
xHigherPriorityTaskWoken = pdFALSE;

// Loop until the buffer is empty.
do
{
// Obtain a byte from the buffer.
cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

// Post the byte.
xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
// Actual macro used here is port specific.
portYIELD_FROM_ISR ();
}
}
```

**Return** `pdTRUE` if the data was successfully sent to the queue, otherwise `errQUEUE_FULL`.

**Parameters**

- `xQueue`: The handle to the queue on which the item is to be posted.
- `pvItemToQueue`: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.
- `pxHigherPriorityTaskWoken`: `xQueueSendFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueSendFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited.

**xQueueReset** (xQueue)

Reset a queue back to its original empty state. `pdPASS` is returned if the queue is successfully reset. `pdFAIL` is returned if the queue could not be reset because there are tasks blocked on the queue waiting to either receive from the queue or send to the queue.

**Return** always returns pdPASS

**Parameters**

- xQueue: The queue to reset

## Type Definitions

**typedef** void \*QueueHandle\_t

Type by which queues are referenced. For example, a call to xQueueCreate() returns an QueueHandle\_t variable that can then be used as a parameter to xQueueSend(), xQueueReceive(), etc.

**typedef** void \*QueueSetHandle\_t

Type by which queue sets are referenced. For example, a call to xQueueCreateSet() returns an xQueueSet variable that can then be used as a parameter to xQueueSelectFromSet(), xQueueAddToSet(), etc.

**typedef** void \*QueueSetMemberHandle\_t

Queue sets can contain both queues and semaphores, so the QueueSetMemberHandle\_t is defined as a type to be used where a parameter or return value can be either an QueueHandle\_t or an SemaphoreHandle\_t.

## Semaphore API

### Header File

- [freertos/include/freertos/semphr.h](#)

### Macros

**semBINARY\_SEMAPHORE\_QUEUE\_LENGTH**

**semSEMAPHORE\_QUEUE\_ITEM\_LENGTH**

**semGIVE\_BLOCK\_TIME**

**xSemaphoreCreateBinary** ()

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see <http://www.freertos.org/a00111.html>). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

The old vSemaphoreCreateBinary() macro is now deprecated in favour of this xSemaphoreCreateBinary() function. Note that binary semaphores created using the vSemaphoreCreateBinary() macro are created in a state such that the first call to 'take' the semaphore would pass, whereas binary semaphores created using xSemaphoreCreateBinary() are created in a state such that the the semaphore must first be 'given' before it can be 'taken'.

Function that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as nothing is actually stored - all that is important is whether the queue is empty or full (the binary semaphore is available or not).

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously ‘give’ the semaphore while another continuously ‘takes’ the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see `xSemaphoreCreateMutex()`.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to vSemaphoreCreateBinary ().
    // This is a macro so pass the variable in directly.
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

**Return** Handle to the created semaphore.

#### **xSemaphoreCreateBinaryStatic** (pxStaticSemaphore)

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

NOTE: In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using `xSemaphoreCreateBinary()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateBinary()` function. (see <http://www.freertos.org/a00111.html>). If a binary semaphore is created using `xSemaphoreCreateBinaryStatic()` then the application writer must provide the memory. `xSemaphoreCreateBinaryStatic()` therefore allows a binary semaphore to be created without using any dynamic memory allocation.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously ‘give’ the semaphore while another continuously ‘takes’ the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see `xSemaphoreCreateMutex()`.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to xSemaphoreCreateBinary().
    // The semaphore's data structures will be placed in the xSemaphoreBuffer
    // variable, the address of which is passed into the function. The
    // function's parameter is not NULL, so the function will not attempt any
    // dynamic memory allocation, and therefore the function will not return
    // return NULL.
    xSemaphore = xSemaphoreCreateBinary( &xSemaphoreBuffer );
```

(continues on next page)

(continued from previous page)

```

// Rest of task code goes here.
}

```

**Return** If the semaphore is created then a handle to the created semaphore is returned. If pxSemaphoreBuffer is NULL then NULL is returned.

#### Parameters

- pxStaticSemaphore: Must point to a variable of type StaticSemaphore\_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically.

#### **xSemaphoreTake** (xSemaphore, xBlockTime)

*Macro* to obtain a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

Example usage:

```

SemaphoreHandle_t xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
    // Create the semaphore to guard a shared resource.
    vSemaphoreCreateBinary( xSemaphore );
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xSemaphore != NULL )
    {
        // See if we can obtain the semaphore. If the semaphore is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
        {
            // We were able to obtain the semaphore and can now access the
            // shared resource.

            // ...

            // We have finished accessing the shared resource. Release the
            // semaphore.
            xSemaphoreGive( xSemaphore );
        }
        else
        {
            // We could not obtain the semaphore and can therefore not access
            // the shared resource safely.
        }
    }
}

```

**Return** pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore

becoming available.

### Parameters

- `xSemaphore`: A handle to the semaphore being taken - obtained when the semaphore was created.
- `xBlockTime`: The time in ticks to wait for the semaphore to become available. The macro `portTICK_PERIOD_MS` can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. A block time of `portMAX_DELAY` can be used to block indefinitely (provided `INCLUDE_vTaskSuspend` is set to 1 in `FreeRTOSConfig.h`).

### `xSemaphoreTakeRecursive` (`xMutex`, `xBlockTime`)

*Macro* to recursively obtain, or 'take', a mutex type semaphore. The mutex must have previously been created using a call to `xSemaphoreCreateRecursiveMutex()`;

`configUSE_RECURSIVE_MUTEXES` must be set to 1 in `FreeRTOSConfig.h` for this macro to be available.

This macro must not be used on mutexes created using `xSemaphoreCreateMutex()`.

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called `xSemaphoreGiveRecursive()` for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xMutex != NULL )
    {
        // 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

```

(continues on next page)

(continued from previous page)

```

    // three times. Again it is unlikely that real code would have
    // these calls sequentially, but instead buried in a more complex
    // call structure. This is just for illustrative purposes.
    xSemaphoreGiveRecursive( xMutex );
    xSemaphoreGiveRecursive( xMutex );
    xSemaphoreGiveRecursive( xMutex );

    // Now the mutex can be taken by other tasks.
}
else
{
    // We could not obtain the mutex and can therefore not access
    // the shared resource safely.
}
}
}

```

**Return** pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

#### Parameters

- xMutex: A handle to the mutex being obtained. This is the handle returned by xSemaphoreCreateRecursiveMutex();
- xBlockTime: The time in ticks to wait for the semaphore to become available. The macro portTICK\_PERIOD\_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then xSemaphoreTakeRecursive() will return immediately no matter what the value of xBlockTime.

#### **xSemaphoreGive** (xSemaphore)

*Macro* to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphoreTake().

This macro must not be used from an ISR. See xSemaphoreGiveFromISR () for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Example usage:

```

SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{
    // Create the semaphore to guard a shared resource.
    vSemaphoreCreateBinary( xSemaphore );

    if( xSemaphore != NULL )
    {
        if( xSemaphoreGive( xSemaphore ) != pdTRUE )
        {
            // We would expect this call to fail because we cannot give
            // a semaphore without first "taking" it!
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

// Obtain the semaphore - don't block if the semaphore is not
// immediately available.
if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
{
    // We now have the semaphore and can access the shared resource.

    // ...

    // We have finished accessing the shared resource so can free the
    // semaphore.
    if( xSemaphoreGive( xSemaphore ) != pdTRUE )
    {
        // We would not expect this call to fail because we must have
        // obtained the semaphore to get here.
    }
}
}
}

```

**Return** pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

#### Parameters

- **xSemaphore**: A handle to the semaphore being released. This is the handle returned when the semaphore was created.

#### **xSemaphoreGiveRecursive**(xMutex)

*Macro* to recursively release, or ‘give’, a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE\_RECURSIVE\_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called xSemaphoreGiveRecursive() for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

Example usage:

```

SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.
}

```

(continues on next page)

(continued from previous page)

```

if( xMutex != NULL )
{
    // See if we can obtain the mutex.  If the mutex is not available
    // wait 10 ticks to see if it becomes free.
    if( xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 ) == pdTRUE )
    {
        // We were able to obtain the mutex and can now access the
        // shared resource.

        // ...
        // For some reason due to the nature of the code further calls to
        // xSemaphoreTakeRecursive() are made on the same mutex.  In real
        // code these would not be just sequential calls as this would make
        // no sense.  Instead the calls are likely to be buried inside
        // a more complex call structure.
        xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
        xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

        // The mutex has now been 'taken' three times, so will not be
        // available to another task until it has also been given back
        // three times.  Again it is unlikely that real code would have
        // these calls sequentially, it would be more likely that the calls
        // to xSemaphoreGiveRecursive() would be called as a call stack
        // unwound.  This is just for demonstrative purposes.
        xSemaphoreGiveRecursive( xMutex );
        xSemaphoreGiveRecursive( xMutex );
        xSemaphoreGiveRecursive( xMutex );

        // Now the mutex can be taken by other tasks.
    }
    else
    {
        // We could not obtain the mutex and can therefore not access
        // the shared resource safely.
    }
}
}

```

**Return** pdTRUE if the semaphore was given.

#### Parameters

- xMutex: A handle to the mutex being released, or 'given'. This is the handle returned by xSemaphoreCreateMutex();

#### **xSemaphoreGiveFromISR** (xSemaphore, pxHigherPriorityTaskWoken)

*Macro* to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR.

Example usage:

```
\#define LONG_TIME 0xffff
\#define TICKS_TO_WAIT 10
SemaphoreHandle_t xSemaphore = NULL;

// Repetitive task.
void vATask( void * pvParameters )
{
    for( ;; )
    {
        // We want this task to run every 10 ticks of a timer.  The semaphore
        // was created before this task was started.

        // Block waiting for the semaphore to become available.
        if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
        {
            // It is time to execute.

            // ...

            // We have finished our task.  Return to the top of the loop where
            // we will block on the semaphore until it is time to execute
            // again.  Note when using the semaphore for synchronisation with an
            // ISR in this manner there is no need to 'give' the semaphore back.
        }
    }
}

// Timer ISR
void vTimerISR( void * pvParameters )
{
    static uint8_t ucLocalTickCount = 0;
    static BaseType_t xHigherPriorityTaskWoken;

    // A timer tick has occurred.

    // ... Do other time functions.

    // Is it time for vATask () to run?
    xHigherPriorityTaskWoken = pdFALSE;
    ucLocalTickCount++;
    if( ucLocalTickCount >= TICKS_TO_WAIT )
    {
        // Unblock the task by releasing the semaphore.
        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

        // Reset the count so we release the semaphore again in 10 ticks time.
        ucLocalTickCount = 0;
    }

    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // We can force a context switch here.  Context switching from an
        // ISR uses port specific syntax.  Check the demo task for your port
        // to find the syntax required.
    }
}
```

**Return** pdTRUE if the semaphore was successfully given, otherwise errQUEUE\_FULL.

#### Parameters

- `xSemaphore`: A handle to the semaphore being released. This is the handle returned when the semaphore was created.
- `pxHigherPriorityTaskWoken`: `xSemaphoreGiveFromISR()` will set `*pxHigherPriorityTaskWoken` to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xSemaphoreGiveFromISR()` sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

#### **xSemaphoreTakeFromISR** (`xSemaphore`, `pxHigherPriorityTaskWoken`)

*Macro* to take a semaphore from an ISR. The semaphore must have previously been created with a call to `vSemaphoreCreateBinary()` or `xSemaphoreCreateCounting()`.

Mutex type semaphores (those created using a call to `xSemaphoreCreateMutex()`) must not be used with this macro.

This macro can be used from an ISR, however taking a semaphore from an ISR is not a common operation. It is likely to only be useful when taking a counting semaphore when an interrupt is obtaining an object from a resource pool (when the semaphore count indicates the number of resources available).

**Return** pdTRUE if the semaphore was successfully taken, otherwise pdFALSE

#### Parameters

- `xSemaphore`: A handle to the semaphore being taken. This is the handle returned when the semaphore was created.
- `pxHigherPriorityTaskWoken`: `xSemaphoreTakeFromISR()` will set `*pxHigherPriorityTaskWoken` to pdTRUE if taking the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xSemaphoreTakeFromISR()` sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

#### **xSemaphoreCreateMutex** ()

*Macro* that implements a mutex semaphore by using the existing queue mechanism.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using `xSemaphoreCreateMutex()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateMutex()` function. (see <http://www.freertos.org/a00111.html>). If a mutex is created using `xSemaphoreCreateMutexStatic()` then the application writer must provide the memory. `xSemaphoreCreateMutexStatic()` therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the `xSemaphoreTake()` and `xSemaphoreGive()` macros. The `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See `vSemaphoreCreateBinary()` for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```

SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
    // This is a macro so pass the variable in directly.
    xSemaphore = xSemaphoreCreateMutex();

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}

```

**Return** If the mutex was successfully created then a handle to the created semaphore is returned. If there was not enough heap to allocate the mutex data structures then NULL is returned.

#### **xSemaphoreCreateMutexStatic** (pxMutexBuffer)

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see <http://www.freertos.org/a00111.html>). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provide the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

Mutexes created using this function can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros must not be used.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```

SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
    // A mutex cannot be used before it has been created. xMutexBuffer is
    // into xSemaphoreCreateMutexStatic() so no dynamic memory allocation is
    // attempted.
    xSemaphore = xSemaphoreCreateMutexStatic( &xMutexBuffer );

    // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
    // so there is no need to check it.
}

```

**Return** If the mutex was successfully created then a handle to the created mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

## Parameters

- `pxMutexBuffer`: Must point to a variable of type `StaticSemaphore_t`, which will be used to hold the mutex's data structure, removing the need for the memory to be allocated dynamically.

### **xSemaphoreCreateRecursiveMutex** ( )

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexes use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using `xSemaphoreCreateRecursiveMutex()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateRecursiveMutex()` function. (see <http://www.freertos.org/a00111.html>). If a recursive mutex is created using `xSemaphoreCreateRecursiveMutexStatic()` then the application writer must provide the memory that will get used by the mutex. `xSemaphoreCreateRecursiveMutexStatic()` therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` macros. The `xSemaphoreTake()` and `xSemaphoreGive()` macros must not be used.

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called `xSemaphoreGiveRecursive()` for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See `vSemaphoreCreateBinary()` for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
    // This is a macro so pass the variable in directly.
    xSemaphore = xSemaphoreCreateRecursiveMutex();

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

**Return** xSemaphore Handle to the created mutex semaphore. Should be of type `SemaphoreHandle_t`.

### **xSemaphoreCreateRecursiveMutexStatic** (pxStaticSemaphore)

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexes use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using `xSemaphoreCreateRecursiveMutex()` then the required

memory is automatically dynamically allocated inside the `xSemaphoreCreateRecursiveMutex()` function. (see <http://www.freertos.org/a00111.html>). If a recursive mutex is created using `xSemaphoreCreateRecursiveMutexStatic()` then the application writer must provide the memory that will get used by the mutex. `xSemaphoreCreateRecursiveMutexStatic()` therefore allows a recursive mutex to be created without using any dynamic memory allocation.

Mutexes created using this macro can be accessed using the `xSemaphoreTakeRecursive()` and `xSemaphoreGiveRecursive()` macros. The `xSemaphoreTake()` and `xSemaphoreGive()` macros must not be used.

A mutex used recursively can be ‘taken’ repeatedly by the owner. The mutex doesn’t become available again until the owner has called `xSemaphoreGiveRecursive()` for each successful ‘take’ request. For example, if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task ‘taking’ a semaphore MUST ALWAYS ‘give’ the semaphore back once the semaphore it is no longer required.

Mutex type semaphores cannot be used from within interrupt service routines.

See `xSemaphoreCreateBinary()` for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always ‘gives’ the semaphore and another always ‘takes’ the semaphore) and from within interrupt service routines.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
    // A recursive semaphore cannot be used before it is created. Here a
    // recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic().
    // The address of xMutexBuffer is passed into the function, and will hold
    // the mutexes data structures - so no dynamic memory allocation will be
    // attempted.
    xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );

    // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
    // so there is no need to check it.
}
```

**Return** If the recursive mutex was successfully created then a handle to the created recursive mutex is returned. If `pxMutexBuffer` was NULL then NULL is returned.

#### Parameters

- `pxStaticSemaphore`: Must point to a variable of type `StaticSemaphore_t`, which will then be used to hold the recursive mutex’s data structure, removing the need for the memory to be allocated dynamically.

#### **xSemaphoreCreateCounting** (uxMaxCount, uxInitialCount)

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using `xSemaphoreCreateCounting()` then the required memory is automatically dynamically allocated inside the `xSemaphoreCreateCounting()`

function. (see <http://www.freertos.org/a00111.html>). If a counting semaphore is created using `xSemaphoreCreateCountingStatic()` then the application writer can instead optionally provide the memory that will get used by the counting semaphore. `xSemaphoreCreateCountingStatic()` therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will ‘give’ a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will ‘take’ a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it ‘gives’ the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    SemaphoreHandle_t xSemaphore = NULL;

    // Semaphore cannot be used before a call to xSemaphoreCreateCounting().
    // The max value to which the semaphore can count should be 10, and the
    // initial value assigned to the count should be 0.
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

**Return** Handle to the created semaphore. Null if the semaphore could not be created.

**Parameters**

- `uxMaxCount`: The maximum count value that can be reached. When the semaphore reaches this value it can no longer be ‘given’.
- `uxInitialCount`: The count value assigned to the semaphore when it is created.

**xSemaphoreCreateCountingStatic** (`uxMaxCount`, `uxInitialCount`, `pxSemaphoreBuffer`)

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! <http://www.freertos.org/RTOS-task-notifications.html>

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using `xSemaphoreCreateCount-`

ing() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see <http://www.freertos.org/a00111.html>). If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the application writer must provide the memory. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will ‘give’ a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will ‘take’ a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it ‘gives’ the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    SemaphoreHandle_t xSemaphore = NULL;

    // Counting semaphore cannot be used before they have been created. Create
    // a counting semaphore using xSemaphoreCreateCountingStatic(). The max
    // value to which the semaphore can count is 10, and the initial value
    // assigned to the count will be 0. The address of xSemaphoreBuffer is
    // passed in and will be used to hold the semaphore structure, so no dynamic
    // memory allocation will be used.
    xSemaphore = xSemaphoreCreateCounting( 10, 0, &xSemaphoreBuffer );

    // No memory allocation was attempted so xSemaphore cannot be NULL, so there
    // is no need to check its value.
}
```

**Return** If the counting semaphore was successfully created then a handle to the created counting semaphore is returned. If pxSemaphoreBuffer was NULL then NULL is returned.

**Parameters**

- uxMaxCount: The maximum count value that can be reached. When the semaphore reaches this value it can no longer be ‘given’.
- uxInitialCount: The count value assigned to the semaphore when it is created.
- pxSemaphoreBuffer: Must point to a variable of type StaticSemaphore\_t, which will then be used to hold the semaphore’s data structure, removing the need for the memory to be allocated dynamically.

**vSemaphoreDelete** (xSemaphore)

Delete a semaphore. This function must be used with care. For example, do not delete a mutex type semaphore

if the mutex is held by a task.

### Parameters

- `xSemaphore`: A handle to the semaphore to be deleted.

#### **xSemaphoreGetMutexHolder** (`xSemaphore`)

If `xMutex` is indeed a mutex type semaphore, return the current mutex holder. If `xMutex` is not a mutex type semaphore, or the mutex is available (not held by a task), return `NULL`.

Note: This is a good way of determining if the calling task is the mutex holder, but not a good way of determining the identity of the mutex holder as the holder may change between the function exiting and the returned value being tested.

#### **uxSemaphoreGetCount** (`xSemaphore`)

If the semaphore is a counting semaphore then `uxSemaphoreGetCount()` returns its current count value. If the semaphore is a binary semaphore then `uxSemaphoreGetCount()` returns 1 if the semaphore is available, and 0 if the semaphore is not available.

## Type Definitions

```
typedef QueueHandle_t SemaphoreHandle_t
```

## Timer API

### Header File

- `freertos/include/freertos/timers.h`

## Functions

```
TimerHandle_t xTimerCreate(const char *const pcTimerName, const TickType_t xTimerPeriod-
    InTicks, const UBaseType_t uxAutoReload, void *const pvTimerID,
    TimerCallbackFunction_t pxCallbackFunction)
```

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using `xTimerCreate()` then the required memory is automatically dynamically allocated inside the `xTimerCreate()` function. (see <http://www.freertos.org/a00111.html>). If a software timer is created using `xTimerCreateStatic()` then the application writer must provide the memory that will get used by the software timer. `xTimerCreateStatic()` therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The `xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` and `xTimerChangePeriodFromISR()` API functions can all be used to transition a timer into the active state.

Example usage:

```
#define NUM_TIMERS 5

// An array to hold handles to the created timers.
TimerHandle_t xTimers[ NUM_TIMERS ];
```

(continues on next page)

(continued from previous page)

```

// An array to hold a count of the number of times each timer expires.
int32_t lExpireCounters[ NUM_TIMERS ] = { 0 };

// Define a callback function that will be used by multiple timer instances.
// The callback function does nothing but count the number of times the
// associated timer expires, and stop the timer once the timer has expired
// 10 times.
void vTimerCallback( TimerHandle_t pxTimer )
{
    int32_t lArrayIndex;
    const int32_t xMaxExpiryCountBeforeStopping = 10;

    // Optionally do something if the pxTimer parameter is NULL.
    configASSERT( pxTimer );

    // Which timer expired?
    lArrayIndex = ( int32_t ) pvTimerGetTimerID( pxTimer );

    // Increment the number of times that pxTimer has expired.
    lExpireCounters[ lArrayIndex ] += 1;

    // If the timer has expired 10 times then stop it from running.
    if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
    {
        // Do not use a block time if calling a timer API function from a
        // timer callback function, as doing so could cause a deadlock!
        xTimerStop( pxTimer, 0 );
    }
}

void main( void )
{
    int32_t x;

    // Create then start some timers. Starting the timers before the scheduler
    // has been started means the timers will start running immediately that
    // the scheduler starts.
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreate( "Timer", // Just a text name, not
        ←used by the kernel.
                                     ( 100 * x ), // The timer period in
        ←ticks.
                                     pdTRUE, // The timers will auto-
        ←reload themselves when they expire.
                                     ( void * ) x, // Assign each timer a
        ←unique id equal to its array index.
                                     vTimerCallback // Each timer calls the
        ←same callback when it expires.
                                     );

        if( xTimers[ x ] == NULL )
        {
            // The timer was not created.
        }
        else

```

(continues on next page)

(continued from previous page)

```

    {
        // Start the timer. No block time is specified, and even if one was
        // it would be ignored because the scheduler has not yet been
        // started.
        if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
        {
            // The timer could not be set into the Active state.
        }
    }
}

// ...
// Create tasks here.
// ...

// Starting the scheduler will start the timers running as they have already
// been set into the active state.
vTaskStartScheduler();

// Should not reach here.
for( ;; );
}

```

**Return** If the timer is successfully created then a handle to the newly created timer is returned. If the timer cannot be created (because either there is insufficient FreeRTOS heap remaining to allocate the timer structures, or the timer period was set to 0) then NULL is returned.

#### Parameters

- `pcTimerName`: A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- `xTimerPeriodInTicks`: The timer period. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then `xTimerPeriodInTicks` should be set to 100. Alternatively, if the timer must expire after 500ms, then `xPeriod` can be set to  $(500 / \text{portTICK\_PERIOD\_MS})$  provided `configTICK_RATE_HZ` is less than or equal to 1000.
- `uxAutoReload`: If `uxAutoReload` is set to `pdTRUE` then the timer will expire repeatedly with a frequency set by the `xTimerPeriodInTicks` parameter. If `uxAutoReload` is set to `pdFALSE` then the timer will be a one-shot timer and enter the dormant state after it expires.
- `pvTimerID`: An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- `pxCallbackFunction`: The function to call when the timer expires. Callback functions must have the prototype defined by `TimerCallbackFunction_t`, which is “void vCallbackFunction( TimerHandle\_t xTimer );”.

*TimerHandle\_t* **xTimerCreateStatic**(const char \*const *pcTimerName*, const TickType\_t *xTimerPeriodInTicks*, const UBaseType\_t *uxAutoReload*, void \*const *pvTimerID*, *TimerCallbackFunction\_t* *pxCallbackFunction*, StaticTimer\_t \**pxTimerBuffer*)

Creates a new software timer instance, and returns a handle by which the created software timer can be referenced.

Internally, within the FreeRTOS implementation, software timers use a block of memory, in which the timer data structure is stored. If a software timer is created using `xTimerCreate()` then the required memory is automatically dynamically allocated inside the `xTimerCreate()` function. (see <http://www.freertos.org/a00111.html>). If a software timer is created using `xTimerCreateStatic()` then the application writer must provide the memory that will get used by the software timer. `xTimerCreateStatic()` therefore allows a software timer to be created without using any dynamic memory allocation.

Timers are created in the dormant state. The `xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` and `xTimerChangePeriodFromISR()` API functions can all be used to transition a timer into the active state.

Example usage:

```
// The buffer used to hold the software timer's data structure.
static StaticTimer_t xTimerBuffer;

// A variable that will be incremented by the software timer's callback
// function.
UBaseType_t uxVariableToIncrement = 0;

// A software timer callback function that increments a variable passed to
// it when the software timer was created. After the 5th increment the
// callback function stops the software timer.
static void prvTimerCallback( TimerHandle_t xExpiredTimer )
{
    UBaseType_t *puxVariableToIncrement;
    BaseType_t xReturned;

    // Obtain the address of the variable to increment from the timer ID.
    puxVariableToIncrement = ( UBaseType_t * ) pvTimerGetTimerID( xExpiredTimer );

    // Increment the variable to show the timer callback has executed.
    ( *puxVariableToIncrement )++;

    // If this callback has executed the required number of times, stop the
    // timer.
    if( *puxVariableToIncrement == 5 )
    {
        // This is called from a timer callback so must not block.
        xTimerStop( xExpiredTimer, staticDONT_BLOCK );
    }
}

void main( void )
{
    // Create the software time. xTimerCreateStatic() has an extra parameter
    // than the normal xTimerCreate() API function. The parameter is a pointer
    // to the StaticTimer_t structure that will hold the software timer
    // structure. If the parameter is passed as NULL then the structure will be
    // allocated dynamically, just as if xTimerCreate() had been called.
    xTimer = xTimerCreateStatic( "T1", // Text name for the task.
    ↪Helps debugging only. Not used by FreeRTOS.
                                xTimerPeriod, // The period of the timer in
    ↪ticks.
                                pdTRUE, // This is an auto-reload
    ↪timer.
                                ( void * ) &uxVariableToIncrement, // A
    ↪variable incremented by the software timer's callback function
```

(continues on next page)

(continued from previous page)

```

                                prvTimerCallback, // The function to execute.
↪when the timer expires.                                &xTimerBuffer ); // The buffer that will hold
↪the software timer structure.

    // The scheduler has not started yet so a block time is not used.
    xReturned = xTimerStart( xTimer, 0 );

    // ...
    // Create tasks here.
    // ...

    // Starting the scheduler will start the timers running as they have already
    // been set into the active state.
    vTaskStartScheduler();

    // Should not reach here.
    for( ;; );
}

```

**Return** If the timer is created then a handle to the created timer is returned. If pxTimerBuffer was NULL then NULL is returned.

#### Parameters

- `pcTimerName`: A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.
- `xTimerPeriodInTicks`: The timer period. The time is defined in tick periods so the constant `portTICK_PERIOD_MS` can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then `xTimerPeriodInTicks` should be set to 100. Alternatively, if the timer must expire after 500ms, then `xPeriod` can be set to  $( 500 / \text{portTICK\_PERIOD\_MS} )$  provided `configTICK_RATE_HZ` is less than or equal to 1000.
- `uxAutoReload`: If `uxAutoReload` is set to `pdTRUE` then the timer will expire repeatedly with a frequency set by the `xTimerPeriodInTicks` parameter. If `uxAutoReload` is set to `pdFALSE` then the timer will be a one-shot timer and enter the dormant state after it expires.
- `pvTimerID`: An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.
- `pxCallbackFunction`: The function to call when the timer expires. Callback functions must have the prototype defined by `TimerCallbackFunction_t`, which is “void vCallbackFunction( TimerHandle\_t xTimer);”.
- `pxTimerBuffer`: Must point to a variable of type `StaticTimer_t`, which will be then be used to hold the software timer’s data structures, removing the need for the memory to be allocated dynamically.

void \***pvTimerGetTimerID**( *TimerHandle\_t xTimer* )

Returns the ID assigned to the timer.

IDs are assigned to timers using the `pvTimerID` parameter of the call to `xTimerCreated()` that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used within the callback function to identify which timer actually expired.

Example usage:

**Return** The ID assigned to the timer being queried.

**Parameters**

- xTimer: The timer being queried.

See the xTimerCreate() API function example usage scenario.

void **vTimerSetTimerID** (*TimerHandle\_t* xTimer, void \*pvNewID)  
Sets the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used as time specific (timer local) storage.

Example usage:

**Parameters**

- xTimer: The timer being updated.
- pvNewID: The ID to assign to the timer.

See the xTimerCreate() API function example usage scenario.

BaseType\_t **xTimerIsTimerActive** (*TimerHandle\_t* xTimer)  
Queries a timer to see if it is active or dormant.

A timer will be dormant if:

- 1) It has been created but **not** started, **or**
- 2) It **is** an expired one-shot timer that has **not** been restarted.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Example usage:

```
// This function assumes xTimer has already been created.
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and_
    →equivalently "if( xTimerIsTimerActive( xTimer ) )"
    {
        // xTimer is active, do something.
    }
    else
    {
        // xTimer is not active, do something else.
    }
}
```

**Return** pdFALSE will be returned if the timer is dormant. A value other than pdFALSE will be returned if the timer is active.

**Parameters**

- xTimer: The timer being queried.

*TaskHandle\_t* **xTimerGetTimerDaemonTaskHandle** (void)

xTimerGetTimerDaemonTaskHandle() is only available if INCLUDE\_xTimerGetTimerDaemonTaskHandle is set to 1 in FreeRTOSConfig.h.

Simply returns the handle of the timer service/daemon task. It is not valid to call xTimerGetTimerDaemonTaskHandle() before the scheduler has been started.

TickType\_t **xTimerGetPeriod** (*TimerHandle\_t* xTimer)

Returns the period of a timer.

**Return** The period of the timer in ticks.

#### Parameters

- xTimer: The handle of the timer being queried.

TickType\_t **xTimerGetExpiryTime** (*TimerHandle\_t* xTimer)

Returns the time in ticks at which the timer will expire. If this is less than the current tick count then the expiry time has overflowed from the current time.

**Return** If the timer is running then the time in ticks at which the timer will next expire is returned. If the timer is not running then the return value is undefined.

#### Parameters

- xTimer: The handle of the timer being queried.

BaseType\_t **xTimerPendFunctionCallFromISR** (*PendedFunction\_t* xFunctionToPend, void \*pvParameter1, uint32\_t ulParameter2, BaseType\_t \*pxHigherPriorityTaskWoken)

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases xTimerPendFunctionCallFromISR() can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the 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:

```
// The callback function that will execute in the context of the daemon task.
// Note callback functions must all use this same prototype.
void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )
{
    BaseType_t xInterfaceToService;

    // The interface that requires servicing is passed in the second
    // parameter. The first parameter is not used in this case.
    xInterfaceToService = ( BaseType_t ) ulParameter2;

    // ...Perform the processing here...
}

// An ISR that receives data packets from multiple interfaces
void vAnISR( void )
{
```

(continues on next page)

(continued from previous page)

```

BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;

// Query the hardware to determine which interface needs processing.
xInterfaceToService = prvCheckInterfaces();

// The actual processing is to be deferred to a task. Request the
// vProcessInterface() callback function is executed, passing in the
// number of the interface that needs processing. The interface to
// service is passed in the second parameter. The first parameter is
// not used in this case.
xHigherPriorityTaskWoken = pdFALSE;
xTimerPendFunctionCallFromISR( vProcessInterface, NULL, ( uint32_t )_
↪xInterfaceToService, &xHigherPriorityTaskWoken );

// If xHigherPriorityTaskWoken is now set to pdTRUE then a context
// switch should be requested. The macro used is port specific and will
// be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
// the documentation page for the port being used.
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

```

**Return** pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

#### Parameters

- xFunctionToPend: The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction\_t prototype.
- pvParameter1: The value of the callback function's first parameter. The parameter has a void \* type to allow it to be used to pass any type. For example, unsigned longs can be cast to a void \*, or the void \* can be used to point to a structure.
- ulParameter2: The value of the callback function's second parameter.
- pxHigherPriorityTaskWoken: As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task (which is set using configTIMER\_TASK\_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then \*pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason \*pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

BaseType\_t **xTimerPendFunctionCall** (*PendedFunction\_t* xFunctionToPend, void \*pvParameter1, uint32\_t ulParameter2, TickType\_t xTicksToWait)

Used to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

**Return** pdPASS is returned if the message was successfully sent to the timer daemon task, otherwise pdFALSE is returned.

#### Parameters

- xFunctionToPend: The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction\_t prototype.

- `pvParameter1`: The value of the callback function's first parameter. The parameter has a `void *` type to allow it to be used to pass any type. For example, unsigned longs can be cast to a `void *`, or the `void *` can be used to point to a structure.
- `ulParameter2`: The value of the callback function's second parameter.
- `xTicksToWait`: Calling this function will result in a message being sent to the timer daemon task on a queue. `xTicksToWait` is the amount of time the calling task should remain in the Blocked state (so not using any processing time) for space to become available on the timer queue if the queue is found to be full.

**const** char \***pcTimerGetTimerName** (*TimerHandle\_t* xTimer)

Returns the name that was assigned to a timer when the timer was created.

**Return** The name assigned to the timer specified by the xTimer parameter.

#### Parameters

- `xTimer`: The handle of the timer being queried.

## Macros

**tmrCOMMAND\_EXECUTE\_CALLBACK\_FROM\_ISR**

**tmrCOMMAND\_EXECUTE\_CALLBACK**

**tmrCOMMAND\_START\_DONT\_TRACE**

**tmrCOMMAND\_START**

**tmrCOMMAND\_RESET**

**tmrCOMMAND\_STOP**

**tmrCOMMAND\_CHANGE\_PERIOD**

**tmrCOMMAND\_DELETE**

**tmrFIRST\_FROM\_ISR\_COMMAND**

**tmrCOMMAND\_START\_FROM\_ISR**

**tmrCOMMAND\_RESET\_FROM\_ISR**

**tmrCOMMAND\_STOP\_FROM\_ISR**

**tmrCOMMAND\_CHANGE\_PERIOD\_FROM\_ISR**

**xTimerStart** (xTimer, xTicksToWait)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerStart()` starts a timer that was previously created using the `xTimerCreate()` API function. If the timer had already been started and was already in the active state, then `xTimerStart()` has equivalent functionality to the `xTimerReset()` API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after `xTimerStart()` was called, where 'n' is the timer's defined period.

It is valid to call `xTimerStart()` before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when `xTimerStart()` was called.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerStart()` to be available.

Example usage:

**Return** `pdFAIL` will be returned if the start command could not be sent to the timer command queue even after `xTicksToWait` ticks had passed. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when `xTimerStart()` is actually called. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

#### Parameters

- `xTimer`: The handle of the timer being started/restarted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when `xTimerStart()` was called. `xTicksToWait` is ignored if `xTimerStart()` is called before the scheduler is started.

See the `xTimerCreate()` API function example usage scenario.

#### **xTimerStop** (`xTimer`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerStop()` stops a timer that was previously started using either of the `xTimerStart()`, `xTimerReset()`, `xTimerStartFromISR()`, `xTimerResetFromISR()`, `xTimerChangePeriod()` or `xTimerChangePeriodFromISR()` API functions.

Stopping a timer ensures the timer is not in the active state.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerStop()` to be available.

Example usage:

**Return** `pdFAIL` will be returned if the stop command could not be sent to the timer command queue even after `xTicksToWait` ticks had passed. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

#### Parameters

- `xTimer`: The handle of the timer being stopped.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when `xTimerStop()` was called. `xTicksToWait` is ignored if `xTimerStop()` is called before the scheduler is started.

See the `xTimerCreate()` API function example usage scenario.

#### **xTimerChangePeriod** (`xTimer`, `xNewPeriod`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer

command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER\_QUEUE\_LENGTH configuration constant.

xTimerChangePeriod() changes the period of a timer that was previously created using the xTimerCreate() API function.

xTimerChangePeriod() can be called to change the period of an active or dormant state timer.

The configUSE\_TIMERS configuration constant must be set to 1 for xTimerChangePeriod() to be available.

Example usage:

```
// This function assumes xTimer has already been created.  If the timer
// referenced by xTimer is already active when it is called, then the timer
// is deleted.  If the timer referenced by xTimer is not active when it is
// called, then the period of the timer is set to 500ms and the timer is
// started.
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and_
    ↪equivalently "if( xTimerIsTimerActive( xTimer ) )"
    {
        // xTimer is already active - delete it.
        xTimerDelete( xTimer );
    }
    else
    {
        // xTimer is not active, change its period to 500ms.  This will also
        // cause the timer to start.  Block for a maximum of 100 ticks if the
        // change period command cannot immediately be sent to the timer
        // command queue.
        ↪if( xTimerChangePeriod( xTimer, 500 / portTICK_PERIOD_MS, 100 ) == pdPASS_
        ↪)
        {
            // The command was successfully sent.
        }
        else
        {
            // The command could not be sent, even after waiting for 100 ticks
            // to pass.  Take appropriate action here.
        }
    }
}
```

**Return** pdFAIL will be returned if the change period command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER\_TASK\_PRIORITY configuration constant.

#### Parameters

- xTimer: The handle of the timer that is having its period changed.
- xNewPeriod: The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK\_PERIOD\_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to ( 500 / portTICK\_PERIOD\_MS ) provided configTICK\_RATE\_HZ is less than or equal to 1000.

- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when `xTimerChangePeriod()` was called. `xTicksToWait` is ignored if `xTimerChangePeriod()` is called before the scheduler is started.

**xTimerDelete** (`xTimer`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerDelete()` deletes a timer that was previously created using the `xTimerCreate()` API function.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerDelete()` to be available.

Example usage:

**Return** `pdFAIL` will be returned if the delete command could not be sent to the timer command queue even after `xTicksToWait` ticks had passed. `pdPASS` will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the `configTIMER_TASK_PRIORITY` configuration constant.

**Parameters**

- `xTimer`: The handle of the timer being deleted.
- `xTicksToWait`: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when `xTimerDelete()` was called. `xTicksToWait` is ignored if `xTimerDelete()` is called before the scheduler is started.

See the `xTimerChangePeriod()` API function example usage scenario.

**xTimerReset** (`xTimer`, `xTicksToWait`)

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` configuration constant.

`xTimerReset()` re-starts a timer that was previously created using the `xTimerCreate()` API function. If the timer had already been started and was already in the active state, then `xTimerReset()` will cause the timer to re-evaluate its expiry time so that it is relative to when `xTimerReset()` was called. If the timer was in the dormant state then `xTimerReset()` has equivalent functionality to the `xTimerStart()` API function.

Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after `xTimerReset()` was called, where 'n' is the timers defined period.

It is valid to call `xTimerReset()` before the scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the scheduler is started, not relative to when `xTimerReset()` was called.

The `configUSE_TIMERS` configuration constant must be set to 1 for `xTimerReset()` to be available.

Example usage:

```
// When a key is pressed, an LCD back-light is switched on. If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off. In
// this case, the timer is a one-shot timer.
```

(continues on next page)

(continued from previous page)

```

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",           // Just a text_
    ↪name, not used by the kernel.                               ( 5000 / portTICK_PERIOD_MS), // The timer_
    ↪period in ticks.                                           pdFALSE,                   // The timer is a_
    ↪one-shot timer.                                             0,                         // The id is not_
    ↪used by the callback so can take any value.                vBacklightTimerCallback   // The callback_
    ↪function that switches the LCD back-light off.              );

    if( xBacklightTimer == NULL )
    {
        // The timer was not created.
    }
    else
    {
        // Start the timer. No block time is specified, and even if one was
        // it would be ignored because the scheduler has not yet been
        // started.
        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
        {

```

(continues on next page)

(continued from previous page)

```

        // The timer could not be set into the Active state.
    }
}

// ...
// Create tasks here.
// ...

// Starting the scheduler will start the timer running as it has already
// been set into the active state.
xTaskStartScheduler();

// Should not reach here.
for( ;; );
}

```

**Return** pdFAIL will be returned if the reset command could not be sent to the timer command queue even after xTicksToWait ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER\_TASK\_PRIORITY configuration constant.

#### Parameters

- xTimer: The handle of the timer being reset/started/restarted.
- xTicksToWait: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when xTimerReset() was called. xTicksToWait is ignored if xTimerReset() is called before the scheduler is started.

#### **xTimerStartFromISR** (xTimer, pxHigherPriorityTaskWoken)

A version of xTimerStart() that can be called from an interrupt service routine.

Example usage:

```

// This scenario assumes xBacklightTimer has already been created. When a
// key is pressed, an LCD back-light is switched on. If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off. In
// this case, the timer is a one-shot timer, and unlike the example given for
// the xTimerReset() function, the key press event handler is an interrupt
// service routine.

// The callback function assigned to the one-shot timer. In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed. Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press interrupt service routine.
void vKeyPressEventInterruptHandler( void )
{

```

(continues on next page)

(continued from previous page)

```

BaseType_t xHigherPriorityTaskWoken = pdFALSE;

// Ensure the LCD back-light is on, then restart the timer that is
// responsible for turning the back-light off after 5 seconds of
// key inactivity. This is an interrupt service routine so can only
// call FreeRTOS API functions that end in "FromISR".
vSetBacklightState( BACKLIGHT_ON );

// xTimerStartFromISR() or xTimerResetFromISR() could be called here
// as both cause the timer to re-calculate its expiry time.
// xHigherPriorityTaskWoken was initialised to pdFALSE when it was
// declared (in this function).
if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=
↳pdPASS )
{
    // The start command was not executed successfully. Take appropriate
    // action here.
}

// Perform the rest of the key processing here.

// If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
// should be performed. The syntax required to perform a context switch
// from inside an ISR varies from port to port, and from compiler to
// compiler. Inspect the demos for the port you are using to find the
// actual syntax required.
if( xHigherPriorityTaskWoken != pdFALSE )
{
    // Call the interrupt safe yield function here (actual function
    // depends on the FreeRTOS port being used).
}
}

```

**Return** pdFAIL will be returned if the start command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStartFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER\_TASK\_PRIORITY configuration constant.

### Parameters

- xTimer: The handle of the timer being started/restarted.
- pxHigherPriorityTaskWoken: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStartFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStartFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then \*pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

**xTimerStopFromISR**( xTimer, pxHigherPriorityTaskWoken )

A version of xTimerStop() that can be called from an interrupt service routine.

Example usage:

```

// This scenario assumes xTimer has already been created and started.  When
// an interrupt occurs, the timer should be simply stopped.

// The interrupt service routine that stops the timer.
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // The interrupt has occurred - simply stop the timer.
    // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    // (within this function).  As this is an interrupt service routine, only
    // FreeRTOS API functions that end in "FromISR" can be used.
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        // The stop command was not executed successfully.  Take appropriate
        // action here.
    }

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed.  The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler.  Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}

```

**Return** pdFAIL will be returned if the stop command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER\_TASK\_PRIORITY configuration constant.

#### Parameters

- xTimer: The handle of the timer being stopped.
- pxHigherPriorityTaskWoken: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStopFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStopFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then \*pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

#### **xTimerChangePeriodFromISR**( xTimer, xNewPeriod, pxHigherPriorityTaskWoken )

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Example usage:

```

// This scenario assumes xTimer has already been created and started.  When
// an interrupt occurs, the period of xTimer should be changed to 500ms.

```

(continues on next page)

(continued from previous page)

```

// The interrupt service routine that changes the period of xTimer.
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // The interrupt has occurred - change the period of xTimer to 500ms.
    // xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    // (within this function). As this is an interrupt service routine, only
    // FreeRTOS API functions that end in "FromISR" can be used.
    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        // The command to change the timers period was not executed
        // successfully. Take appropriate action here.
    }

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed. The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler. Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}

```

**Return** pdFAIL will be returned if the command to change the timers period could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER\_TASK\_PRIORITY configuration constant.

#### Parameters

- xTimer: The handle of the timer that is having its period changed.
- xNewPeriod: The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK\_PERIOD\_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to ( 500 / portTICK\_PERIOD\_MS ) provided configTICK\_RATE\_HZ is less than or equal to 1000.
- pxHigherPriorityTaskWoken: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerChangePeriodFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerChangePeriodFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then \*pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If xTimerChangePeriodFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

**xTimerResetFromISR** (xTimer, pxHigherPriorityTaskWoken)

A version of xTimerReset() that can be called from an interrupt service routine.

Example usage:

```
// This scenario assumes xBacklightTimer has already been created. When a
// key is pressed, an LCD back-light is switched on. If 5 seconds pass
// without a key being pressed, then the LCD back-light is switched off. In
// this case, the timer is a one-shot timer, and unlike the example given for
// the xTimerReset() function, the key press event handler is an interrupt
// service routine.

// The callback function assigned to the one-shot timer. In this case the
// parameter is not used.
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    // The timer expired, therefore 5 seconds must have passed since a key
    // was pressed. Switch off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

// The key press interrupt service routine.
void vKeyPressEventInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // Ensure the LCD back-light is on, then reset the timer that is
    // responsible for turning the back-light off after 5 seconds of
    // key inactivity. This is an interrupt service routine so can only
    // call FreeRTOS API functions that end in "FromISR".
    vSetBacklightState( BACKLIGHT_ON );

    // xTimerStartFromISR() or xTimerResetFromISR() could be called here
    // as both cause the timer to re-calculate its expiry time.
    // xHigherPriorityTaskWoken was initialised to pdFALSE when it was
    // declared (in this function).
    if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) !=
↳pdPASS )
    {
        // The reset command was not executed successfully. Take appropriate
        // action here.
    }

    // Perform the rest of the key processing here.

    // If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    // should be performed. The syntax required to perform a context switch
    // from inside an ISR varies from port to port, and from compiler to
    // compiler. Inspect the demos for the port you are using to find the
    // actual syntax required.
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        // Call the interrupt safe yield function here (actual function
        // depends on the FreeRTOS port being used).
    }
}
```

**Return** pdFAIL will be returned if the reset command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerResetFromISR() is actually called. The

timer service/daemon task priority is set by the configTIMER\_TASK\_PRIORITY configuration constant.

### Parameters

- `xTimer`: The handle of the timer that is to be started, reset, or restarted.
- `pxHigherPriorityTaskWoken`: The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling `xTimerResetFromISR()` writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling `xTimerResetFromISR()` causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then `*pxHigherPriorityTaskWoken` will get set to `pdTRUE` internally within the `xTimerResetFromISR()` function. If `xTimerResetFromISR()` sets this value to `pdTRUE` then a context switch should be performed before the interrupt exits.

### Type Definitions

**typedef** void \***TimerHandle\_t**

Type by which software timers are referenced. For example, a call to `xTimerCreate()` returns an `TimerHandle_t` variable that can then be used to reference the subject timer in calls to other software timer API functions (for example, `xTimerStart()`, `xTimerReset()`, etc.).

**typedef** void (\***TimerCallbackFunction\_t**) (*TimerHandle\_t* xTimer)

Defines the prototype to which timer callback functions must conform.

**typedef** void (\***PendedFunction\_t**) (void \*, uint32\_t)

Defines the prototype to which functions used with the `xTimerPendFunctionCallFromISR()` function must conform.

### Event Group API

#### Header File

- `freertos/include/freertos/event_groups.h`

### Functions

*EventGroupHandle\_t* **xEventGroupCreate** (void)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using `xEventGropuCreate()` then the required memory is automatically dynamically allocated inside the `xEventGroupCreate()` function. (see <http://www.freertos.org/a00111.html>). If an event group is created using `xEventGropuCreateStatic()` then the application writer must instead provide the memory that will get used by the event group. `xEventGroupCreateStatic()` therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the `configUSE_16_BIT_TICKS` setting in `FreeRTOSConfig.h`. If `configUSE_16_BIT_TICKS` is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If `configUSE_16_BIT_TICKS` is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The `EventBits_t` type is used to store event bits within an event group.

Example usage:

```

// 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.
}

```

**Return** If the event group was created then a handle to the event group is returned. If there was insufficient FreeRTOS heap available to create the event group then NULL is returned. See <http://www.freertos.org/a00111.html>

*EventGroupHandle\_t* **xEventGroupCreateStatic** (StaticEventGroup\_t \*pxEventGroupBuffer)

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGropuCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see <http://www.freertos.org/a00111.html>). If an event group is created using xEventGropuCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE\_16\_BIT\_TICKS setting in FreeRTOSConfig.h. If configUSE\_16\_BIT\_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE\_16\_BIT\_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits\_t type is used to store event bits within an event group.

Example usage:

```

// StaticEventGroup_t is a publicly accessible structure that has the same
// size and alignment requirements as the real event group structure. It is
// provided as a mechanism for applications to know the size of the event
// group (which is dependent on the architecture and configuration file
// settings) without breaking the strict data hiding policy by exposing the
// real event group internals. This StaticEventGroup_t variable is passed
// into the xSemaphoreCreateEventGroupStatic() function and is used to store
// the event group's data structures
StaticEventGroup_t xEventGroupBuffer;

// Create the event group without dynamically allocating any memory.
xEventGroup = xEventGroupCreateStatic( &xEventGroupBuffer );

```

**Return** If the event group was created then a handle to the event group is returned. If pxEventGroupBuffer was NULL then NULL is returned.

**Parameters**

- `pxEventGroupBuffer`: `pxEventGroupBuffer` must point to a variable of type `StaticEventGroup_t`, which will be then be used to hold the event group's data structures, removing the need for the memory to be allocated dynamically.

*EventBits\_t* **xEventGroupWaitBits**(*EventGroupHandle\_t* xEventGroup, **const** *EventBits\_t* uxBitsToWaitFor, **const** BaseType\_t xClearOnExit, **const** BaseType\_t xWaitForAllBits, TickType\_t xTicksToWait)

[Potentially] block to wait for one or more bits to be set within a previously created event group.

This function cannot be called from an interrupt.

Example usage:

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
    EventBits_t uxBits;
    const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    // Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
    // the event group. Clear the bits before exiting.
    uxBits = xEventGroupWaitBits(
        xEventGroup,    // The event group being tested.
        BIT_0 | BIT_4, // The bits within the event group to wait for.
        pdTRUE,        // BIT_0 and BIT_4 should be cleared before_
        ↪returning.
        pdFALSE,       // Don't wait for both bits, either bit will do.
        xTicksToWait ); // Wait a maximum of 100ms for either bit to be_
        ↪set.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // xEventGroupWaitBits() returned because both bits were set.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_0 was set.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // xEventGroupWaitBits() returned because just BIT_4 was set.
    }
    else
    {
        // xEventGroupWaitBits() returned because xTicksToWait ticks passed
        // without either BIT_0 or BIT_4 becoming set.
    }
}
```

{c}

**Return** The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If `xEventGroupWaitBits()` returned because its timeout expired then not all the bits being waited for will be set. If `xEventGroupWaitBits()` returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared in the case that `xClearOnExit` parameter was set to `pdTRUE`.

## Parameters

- `xEventGroup`: The event group in which the bits are being tested. The event group must have previously been created using a call to `xEventGroupCreate()`.
- `uxBitsToWaitFor`: A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set `uxBitsToWaitFor` to `0x05`. To wait for bits 0 and/or bit 1 and/or bit 2 set `uxBitsToWaitFor` to `0x07`. Etc.
- `xClearOnExit`: If `xClearOnExit` is set to `pdTRUE` then any bits within `uxBitsToWaitFor` that are set within the event group will be cleared before `xEventGroupWaitBits()` returns if the wait condition was met (if the function returns for a reason other than a timeout). If `xClearOnExit` is set to `pdFALSE` then the bits set in the event group are not altered when the call to `xEventGroupWaitBits()` returns.
- `xWaitForAllBits`: If `xWaitForAllBits` is set to `pdTRUE` then `xEventGroupWaitBits()` will return when either all the bits in `uxBitsToWaitFor` are set or the specified block time expires. If `xWaitForAllBits` is set to `pdFALSE` then `xEventGroupWaitBits()` will return when any one of the bits set in `uxBitsToWaitFor` is set or the specified block time expires. The block time is specified by the `xTicksToWait` parameter.
- `xTicksToWait`: The maximum amount of time (specified in 'ticks') to wait for one/all (depending on the `xWaitForAllBits` value) of the bits specified by `uxBitsToWaitFor` to become set.

*EventBits\_t* **xEventGroupClearBits** (*EventGroupHandle\_t* *xEventGroup*, **const** *EventBits\_t* *uxBitsToClear*)

Clear bits within an event group. This function cannot be called from an interrupt.

Example usage:

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
    EventBits_t uxBits;

    // Clear bit 0 and bit 4 in xEventGroup.
    uxBits = xEventGroupClearBits(
                xEventGroup,    // The event group being updated.
                BIT_0 | BIT_4 ); // The bits being cleared.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // Both bit 0 and bit 4 were set before xEventGroupClearBits() was
        // called. Both will now be clear (not set).
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 was set before xEventGroupClearBits() was called. It will
        // now be clear.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 was set before xEventGroupClearBits() was called. It will
        // now be clear.
    }
    else
    {
        // Neither bit 0 nor bit 4 were set in the first place.
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

```

**Return** The value of the event group before the specified bits were cleared.

#### Parameters

- `xEventGroup`: The event group in which the bits are to be cleared.
- `uxBitsToClear`: A bitwise value that indicates the bit or bits to clear in the event group. For example, to clear bit 3 only, set `uxBitsToClear` to `0x08`. To clear bit 3 and bit 0 set `uxBitsToClear` to `0x09`.

*EventBits\_t* **xEventGroupSetBits** (*EventGroupHandle\_t* `xEventGroup`, **const** *EventBits\_t* `uxBitsToSet`)

Set bits within an event group. This function cannot be called from an interrupt. `xEventGroupSetBitsFromISR()` is a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

Example usage:

```

#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
    EventBits_t uxBits;

    // Set bit 0 and bit 4 in xEventGroup.
    uxBits = xEventGroupSetBits(
                xEventGroup,    // The event group being updated.
                BIT_0 | BIT_4 ); // The bits being set.

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        // Both bit 0 and bit 4 remained set when the function returned.
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        // Bit 0 remained set when the function returned, but bit 4 was
        // cleared. It might be that bit 4 was cleared automatically as a
        // task that was waiting for bit 4 was removed from the Blocked
        // state.
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        // Bit 4 remained set when the function returned, but bit 0 was
        // cleared. It might be that bit 0 was cleared automatically as a
        // task that was waiting for bit 0 was removed from the Blocked
        // state.
    }
    else
    {
        // Neither bit 0 nor bit 4 remained set. It might be that a task
        // was waiting for both of the bits to be set, and the bits were
        // cleared as the task left the Blocked state.
    }
}

```

```
{c}
```

**Return** The value of the event group at the time the call to `xEventGroupSetBits()` returns. There are two reasons why the returned value might have the bits specified by the `uxBitsToSet` parameter cleared. First, if setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will be cleared automatically (see the `xClearBitOnExit` parameter of `xEventGroupWaitBits()`). Second, any unblocked (or otherwise Ready state) task that has a priority above that of the task that called `xEventGroupSetBits()` will execute and may change the event group value before the call to `xEventGroupSetBits()` returns.

#### Parameters

- `xEventGroup`: The event group in which the bits are to be set.
- `uxBitsToSet`: A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set `uxBitsToSet` to `0x08`. To set bit 3 and bit 0 set `uxBitsToSet` to `0x09`.

*EventBits\_t* **xEventGroupSync** (*EventGroupHandle\_t* `xEventGroup`, **const** *EventBits\_t* `uxBitsToSet`, **const** *EventBits\_t* `uxBitsToWaitFor`, *TickType\_t* `xTicksToWait`)

Atomically set bits within an event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronise multiple tasks, where each task has to wait for the other tasks to reach a synchronisation point before proceeding.

This function cannot be used from an interrupt.

The function will return before its block time expires if the bits specified by the `uxBitsToWait` parameter are set, or become set within that time. In this case all the bits specified by `uxBitsToWait` will be automatically cleared before the function returns.

Example usage:

```
// Bits used by the three tasks.
#define TASK_0_BIT    ( 1 << 0 )
#define TASK_1_BIT    ( 1 << 1 )
#define TASK_2_BIT    ( 1 << 2 )

#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

// Use an event group to synchronise three tasks. It is assumed this event
// group has already been created elsewhere.
EventGroupHandle_t xEventBits;

void vTask0( void *pvParameters )
{
    EventBits_t uxReturn;
    TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    for( ;; )
    {
        // Perform task functionality here.

        // Set bit 0 in the event flag to note this task has reached the
        // sync point. The other two tasks will set the other two bits defined
        // by ALL_SYNC_BITS. All three tasks have reached the synchronisation
        // point when all the ALL_SYNC_BITS are set. Wait a maximum of 100ms
        // for this to happen.
        uxReturn = xEventGroupSync( xEventBits, TASK_0_BIT, ALL_SYNC_BITS,
        ↪xTicksToWait );

        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
```

(continues on next page)

(continued from previous page)

```

        {
            // 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.
    }
}

```

**Return** The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If `xEventGroupSync()` returned because its timeout expired then not all the bits being waited for will be set. If `xEventGroupSync()` returned because all the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared.

#### Parameters

- `xEventGroup`: The event group in which the bits are being tested. The event group must have previously been created using a call to `xEventGroupCreate()`.
- `uxBitsToSet`: The bits to set in the event group before determining if, and possibly waiting for, all

the bits specified by the `uxBitsToWait` parameter are set.

- `uxBitsToWaitFor`: A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and bit 2 set `uxBitsToWaitFor` to `0x05`. To wait for bits 0 and bit 1 and bit 2 set `uxBitsToWaitFor` to `0x07`. Etc.
- `xTicksToWait`: The maximum amount of time (specified in 'ticks') to wait for all of the bits specified by `uxBitsToWaitFor` to become set.

*EventBits\_t* **xEventGroupGetBitsFromISR** (*EventGroupHandle\_t* *xEventGroup*)

A version of `xEventGroupGetBits()` that can be called from an ISR.

**Return** The event group bits at the time `xEventGroupGetBitsFromISR()` was called.

#### Parameters

- `xEventGroup`: The event group being queried.

void **vEventGroupDelete** (*EventGroupHandle\_t* *xEventGroup*)

Delete an event group that was previously created by a call to `xEventGroupCreate()`. Tasks that are blocked on the event group will be unblocked and obtain 0 as the event group's value.

#### Parameters

- `xEventGroup`: The event group being deleted.

## Macros

**xEventGroupClearBitsFromISR** (*xEventGroup*, *uxBitsToClear*)

A version of `xEventGroupClearBits()` that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed while interrupts are disabled, so protects event groups that are accessed from tasks by suspending the scheduler rather than disabling interrupts. As a result event groups cannot be accessed directly from an interrupt service routine. Therefore `xEventGroupClearBitsFromISR()` sends a message to the timer task to have the clear operation performed in the context of the timer task.

Example usage:

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

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.
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

```

**Return** If the request to execute the function was posted successfully then `pdPASS` is returned, otherwise `pdFALSE` is returned. `pdFALSE` will be returned if the timer service queue was full.

#### Parameters

- `xEventGroup`: The event group in which the bits are to be cleared.
- `uxBitsToClear`: A bitwise value that indicates the bit or bits to clear. For example, to clear bit 3 only, set `uxBitsToClear` to `0x08`. To clear bit 3 and bit 0 set `uxBitsToClear` to `0x09`.

#### **xEventGroupSetBitsFromISR** (`xEventGroup`, `uxBitsToSet`, `pxHigherPriorityTaskWoken`)

A version of `xEventGroupSetBits()` that can be called from an interrupt.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow nondeterministic operations to be performed in interrupts or from critical sections. Therefore `xEventGroupSetBitFromISR()` sends a message to the timer task to have the set operation performed in the context of the timer task - where a scheduler lock is used in place of a critical section.

Example usage:

```

#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

// An event group which it is assumed has already been created by a call to
// xEventGroupCreate().
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken, xResult;

    // xHigherPriorityTaskWoken must be initialised to pdFALSE.
    xHigherPriorityTaskWoken = pdFALSE;

    // Set bit 0 and bit 4 in xEventGroup.
    xResult = xEventGroupSetBitsFromISR(
                xEventGroup,    // The event group being updated.
                BIT_0 | BIT_4   // The bits being set.
                &xHigherPriorityTaskWoken );

    // Was the message posted successfully?
    if( xResult == pdPASS )
    {
        // If xHigherPriorityTaskWoken is now set to pdTRUE then a context
        // switch should be requested. The macro used is port specific and
        // will be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() -
        // refer to the documentation page for the port being used.
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
}

```

**Return** If the request to execute the function was posted successfully then `pdPASS` is returned, otherwise `pdFALSE` is returned. `pdFALSE` will be returned if the timer service queue was full.

### Parameters

- `xEventGroup`: The event group in which the bits are to be set.
- `uxBitsToSet`: A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set `uxBitsToSet` to `0x08`. To set bit 3 and bit 0 set `uxBitsToSet` to `0x09`.
- `pxHigherPriorityTaskWoken`: As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then `*pxHigherPriorityTaskWoken` will be set to `pdTRUE` by `xEventGroupSetBitsFromISR()`, indicating that a context switch should be requested before the interrupt exits. For that reason `*pxHigherPriorityTaskWoken` must be initialised to `pdFALSE`. See the example code below.

### `xEventGroupGetBits` (`xEventGroup`)

Returns the current value of the bits in an event group. This function cannot be used from an interrupt.

**Return** The event group bits at the time `xEventGroupGetBits()` was called.

### Parameters

- `xEventGroup`: The event group being queried.

## Type Definitions

```
typedef void *EventGroupHandle_t
```

An event group is a collection of bits to which an application can assign a meaning. For example, an application may create an event group to convey the status of various CAN bus related events in which bit 0 might mean “A CAN

message has been received and is ready for processing”, bit 1 might mean “The application has queued a message that is ready for sending onto the CAN network”, and bit 2 might mean “It is time to send a SYNC message onto the CAN network” etc. A task can then test the bit values to see which events are active, and optionally enter the Blocked state to wait for a specified bit or a group of specified bits to be active. To continue the CAN bus example, a CAN controlling task can enter the Blocked state (and therefore not consume any processing time) until either bit 0, bit 1 or bit 2 are active, at which time the bit that was actually active would inform the task which action it had to take (process a received message, send a message, or send a SYNC).

The event groups implementation contains intelligence to avoid race conditions that would otherwise occur were an application to use a simple variable for the same purpose. This is particularly important with respect to when a bit within an event group is to be cleared, and when bits have to be set and then tested atomically - as is the case where event groups are used to create a synchronisation point between multiple tasks (a ‘rendezvous’).  
`event_groups.h`

Type by which event groups are referenced. For example, a call to `xEventGroupCreate()` returns an `EventGroupHandle_t` variable that can then be used as a parameter to other event group functions.

```
typedef TickType_t EventBits_t
```

## Ringbuffer API

### Header File

- [freertos/include/freertos/ringbuf.h](#)

## Functions

*RingbufHandle\_t* **xRingbufferCreate** (*size\_t buf\_length*, *ringbuf\_type\_t type*)

Create a ring buffer.

**Return** A *RingbufHandle\_t* handle to the created ringbuffer, or NULL in case of error.

### Parameters

- *buf\_length*: Length of circular buffer, in bytes. Each entry will take up its own length, plus a header that at the moment is equal to `sizeof(size_t)`.
- *type*: Type of ring buffer, see *ringbuf\_type\_t*.

*RingbufHandle\_t* **xRingbufferCreateNoSplit** (*size\_t item\_size*, *size\_t num\_item*)

Create a ring buffer of type `RINGBUF_TYPE_NOSPLIT` for a fixed *item\_size*.

This API is similar to `xRingbufferCreate()`, but it will internally allocate additional space for the headers.

**Return** A *RingbufHandle\_t* handle to the created ringbuffer, or NULL in case of error.

### Parameters

- *item\_size*: Size of each item to be put into the ring buffer
- *num\_item*: Maximum number of items the buffer needs to hold simultaneously

void **vRingbufferDelete** (*RingbufHandle\_t ringbuf*)

Delete a ring buffer.

### Parameters

- *ringbuf*: Ring buffer to delete

*size\_t* **xRingbufferGetMaxItemSize** (*RingbufHandle\_t ringbuf*)

Get maximum size of an item that can be placed in the ring buffer.

**Return** Maximum size, in bytes, of an item that can be placed in a ring buffer.

### Parameters

- *ringbuf*: Ring buffer to query

*size\_t* **xRingbufferGetCurFreeSize** (*RingbufHandle\_t ringbuf*)

Get current free size available in the buffer.

This gives the real time free space available in the ring buffer. So basically, this will be the maximum size of the entry that can be sent into the buffer.

**Note** 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`

**Return** Current free size, in bytes, available for an entry

### Parameters

- *ringbuf*: - Ring buffer to query

bool **xRingbufferIsNextItemWrapped** (*RingbufHandle\_t ringbuf*)

Check if the next item is wrapped.

This API tells if the next item that is available for a Receive is wrapped or not. This is valid only if the ring buffer type is RINGBUF\_TYPE\_ALLOWSPPLIT

**Note** 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 Receive

**Return** true if the next item is wrapped around

**Return** false if the next item is not wrapped

**Parameters**

- ringbuf: - Ring buffer to query

BaseType\_t **xRingbufferSend** (*RingbufHandle\_t ringbuf*, void \**data*, size\_t *data\_size*, TickType\_t *ticks\_to\_wait*)

Insert an item into the ring buffer.

**Return**

- pdTRUE if succeeded
- pdFALSE on time-out or when the buffer is larger than indicated by xRingbufferGetMaxItemSize(ringbuf).

**Parameters**

- ringbuf: Ring buffer to insert the item into
- data: Pointer to data to insert. NULL is allowed if data\_size is 0.
- data\_size: Size of data to insert. A value of 0 is allowed.
- ticks\_to\_wait: Ticks to wait for room in the ringbuffer.

BaseType\_t **xRingbufferSendFromISR** (*RingbufHandle\_t ringbuf*, void \**data*, size\_t *data\_size*, BaseType\_t \**higher\_prio\_task\_awoken*)

Insert an item into the ring buffer from an ISR.

**Return** pdTRUE if succeeded, pdFALSE when the ring buffer does not have space.

**Parameters**

- ringbuf: Ring buffer to insert the item into
- data: Pointer to data to insert. NULL is allowed if data\_size is 0.
- data\_size: Size of data to insert. A value of 0 is allowed.
- higher\_prio\_task\_awoken: Value pointed to will be set to pdTRUE if the push woke up a higher priority task.

void \***xRingbufferReceive** (*RingbufHandle\_t ringbuf*, size\_t \**item\_size*, TickType\_t *ticks\_to\_wait*)

Retrieve an item from the ring buffer.

**Note** A call to vRingbufferReturnItem() is required after this to free up the data received.

**Return**

- pointer to the retrieved item on success; \*item\_size filled with the length of the item.

- NULL on timeout, *\*item\_size* is untouched in that case.

**Parameters**

- *ringbuf*: Ring buffer to retrieve the item from
- *item\_size*: Pointer to a variable to which the size of the retrieved item will be written.
- *ticks\_to\_wait*: Ticks to wait for items in the ringbuffer.

void **\*xRingbufferReceiveFromISR** (*RingbufHandle\_t ringbuf*, *size\_t \*item\_size*)  
 Retrieve an item from the ring buffer from an ISR.

**Note** A call to `vRingbufferReturnItemFromISR()` is required after this to free up the data received

**Return**

- Pointer to the retrieved item on success; *\*item\_size* filled with the length of the item.
- NULL when the ringbuffer is empty, *\*item\_size* is untouched in that case.

**Parameters**

- *ringbuf*: Ring buffer to retrieve the item from
- *item\_size*: Pointer to a variable to which the size of the retrieved item will be written.

void **\*xRingbufferReceiveUpTo** (*RingbufHandle\_t ringbuf*, *size\_t \*item\_size*, *TickType\_t ticks\_to\_wait*,  
*size\_t wanted\_size*)  
 Retrieve bytes from a ByteBuf type of ring buffer, specifying the maximum amount of bytes to return.

**Note** A call to `vRingbufferReturnItem()` is required after this to free up the data received.

**Return**

- Pointer to the retrieved item on success; *\*item\_size* filled with the length of the item.
- NULL on timeout, *\*item\_size* is untouched in that case.

**Parameters**

- *ringbuf*: Ring buffer to retrieve the item from
- *item\_size*: Pointer to a variable to which the size of the retrieved item will be written.
- *ticks\_to\_wait*: Ticks to wait for items in the ringbuffer.
- *wanted\_size*: Maximum number of bytes to return.

void **\*xRingbufferReceiveUpToFromISR** (*RingbufHandle\_t ringbuf*, *size\_t \*item\_size*, *size\_t*  
*wanted\_size*)  
 Retrieve bytes from a ByteBuf type of ring buffer, specifying the maximum amount of bytes to return. Call this from an ISR.

**Note** A call to `vRingbufferReturnItemFromISR()` is required after this to free up the data received.

**Return**

- Pointer to the retrieved item on success; *\*item\_size* filled with the length of the item.
- NULL when the ringbuffer is empty, *\*item\_size* is untouched in that case.

**Parameters**

- *ringbuf*: Ring buffer to retrieve the item from

- `item_size`: Pointer to a variable to which the size of the retrieved item will be written.
- `wanted_size`: Maximum number of bytes to return.

void **vRingbufferReturnItem** (*RingbufHandle\_t* ringbuf, void \*item)  
Return a previously-retrieved item to the ringbuffer.

#### Parameters

- `ringbuf`: Ring buffer the item was retrieved from
- `item`: Item that was received earlier

void **vRingbufferReturnItemFromISR** (*RingbufHandle\_t* ringbuf, void \*item, BaseType\_t \*higher\_prio\_task\_awoken)  
Return a previously-retrieved item to the ringbuffer from an ISR.

#### Parameters

- `ringbuf`: Ring buffer the item was retrieved from
- `item`: Item that was received earlier
- `higher_prio_task_awoken`: Value pointed to will be set to pdTRUE if the push woke up a higher priority task.

BaseType\_t **xRingbufferAddToQueueSetRead** (*RingbufHandle\_t* ringbuf, *QueueSetHandle\_t* xQueueSet)  
Add the ringbuffer to a queue set.

This specifically adds the semaphore that indicates more space has become available in the ringbuffer.

#### Return

- pdTRUE on success, pdFALSE otherwise

#### Parameters

- `ringbuf`: Ring buffer to add to the queue set
- `xQueueSet`: Queue set to add the ringbuffer to

BaseType\_t **xRingbufferAddToQueueSetWrite** (*RingbufHandle\_t* ringbuf, *QueueSetHandle\_t* xQueueSet)  
Add the ringbuffer to a queue set.

This specifically adds the semaphore that indicates something has been written into the ringbuffer.

**Return** pdTRUE on success, pdFALSE otherwise

#### Parameters

- `ringbuf`: Ring buffer to add to the queue set
- `xQueueSet`: Queue set to add the ringbuffer to

BaseType\_t **xRingbufferRemoveFromQueueSetRead** (*RingbufHandle\_t* ringbuf, *QueueSetHandle\_t* xQueueSet)  
Remove the ringbuffer from a queue set.

This specifically removes the semaphore that indicates more space has become available in the ringbuffer.

**Return** pdTRUE on success, pdFALSE otherwise

**Parameters**

- `ringbuf`: Ring buffer to remove from the queue set
- `xQueueSet`: Queue set to remove the ringbuffer from

BaseType\_t **xRingbufferRemoveFromQueueSetWrite** (*RingbufHandle\_t ringbuf*, *QueueSetHandle\_t xQueueSet*)

Remove the ringbuffer from a queue set.

This specifically removes the semaphore that indicates something has been written to the ringbuffer.

**Return** pdTRUE on success, pdFALSE otherwise

**Parameters**

- `ringbuf`: Ring buffer to remove from the queue set
- `xQueueSet`: Queue set to remove the ringbuffer from

void **xRingbufferPrintInfo** (*RingbufHandle\_t ringbuf*)

Debugging function to print the internal pointers in the ring buffer.

**Parameters**

- `ringbuf`: Ring buffer to show

**Type Definitions**

```
typedef void *RingbufHandle_t
```

**Enumerations**

```
enum ringbuf_type_t
```

The various types of buffer.

A ringbuffer instantiated by these functions essentially acts like a FreeRTOS queue, with the difference that it's strictly FIFO and with the main advantage that you can put in randomly-sized items. The capacity, accordingly, isn't measured in the amount of items, but the amount of memory that is used for storing the items. Dependent on the size of the items, more or less of them will fit in the ring buffer.

This ringbuffer tries to be efficient with memory: when inserting an item, the item data will be copied to the ringbuffer memory. When retrieving an item, however, a reference to ringbuffer memory will be returned. The returned memory is guaranteed to be 32-bit aligned and contiguous. The application can use this memory, but as long as it does, ringbuffer writes that would write to this bit of memory will block.

The requirement for items to be contiguous is slightly problematic when the only way to place the next item would involve a wraparound from the end to the beginning of the ringbuffer. This can be solved (or not) in a few ways, see descriptions of possible `ringbuf_type_t` types below.

The maximum size of an item will be affected by ringbuffer type. When split items are allowed, it is acceptable to push items of  $(\text{buffer\_size}) - 16$  bytes into the buffer. When it's not allowed, the maximum size is  $(\text{buffer\_size}/2) - 8$  bytes. The bytebuf can fill the entire buffer with data, it has no overhead.

*Values:*

```
RINGBUF_TYPE_NOSPLIT = 0
```

The insertion code will leave the room at the end of the ringbuffer unused and instead will put the entire item at the start of the ringbuffer, as soon as there is enough free space.

#### **RINGBUF\_TYPE\_ALLOWSPLIT**

The insertion code will split the item in two items; one which fits in the space left at the end of the ringbuffer, one that contains the remaining data which is placed in the beginning. Two `xRingbufferReceive` calls will be needed to retrieve the data.

#### **RINGBUF\_TYPE\_BYTEBUF**

This is your conventional byte-based ringbuffer. It does have no overhead, but it has no item contiguousness either: a read will just give you the entire written buffer space, or the space up to the end of the buffer, and writes can be broken up in any way possible. Note that this type cannot do a 2nd read before returning the memory of the 1st.

## 2.7.2 FreeRTOS Hooks

### Overview

FreeRTOS consists of Idle Hooks and Tick Hooks which allow for application specific functionality to be added to the Idle Task and Tick Interrupt. The ESP32 is dual core in nature, hence the ESP-IDF provides its own Idle and Tick Hooks that are dual core compatible in addition to the hooks provided by Vanilla FreeRTOS.

### Vanilla FreeRTOS Hooks

Idle and Tick Hooks in vanilla FreeRTOS are implemented by defining implementations for the functions `vApplicationIdleHook` and `vApplicationTickHook` respectively somewhere in the application. Vanilla FreeRTOS will run the user defined Idle Hook every iteration of the Idle Task, whereas the user defined Tick Hook will run once per tick interrupt (given that there are no pended ticks).

Due to vanilla FreeRTOS being designed for single core, `vApplicationIdleHook` and `vApplicationTickHook` will be run in both cores on the ESP32. In other words, the same Idle Hook and Tick Hook are used for both cores.

To enable the vanilla FreeRTOS hooks in ESP-IDF, `FREERTOS_LEGACY_HOOKS` must be enabled in `make menuconfig`. `FREERTOS_LEGACY_IDLE_HOOK` and `FREERTOS_LEGACY_TICK_HOOK` should also be enabled.

### ESP-IDF Idle and Tick Hooks

Due to the dual core nature of the ESP32, it may be necessary for some applications to have separate Idle Hooks for each core. Furthermore, it may be necessary for Idle and Tick Hooks to have execute multiple functionalities that are configurable at run time. Therefore the ESP-IDF provides its own Idle and Tick Hooks in addition to the hooks provided by Vanilla FreeRTOS.

The ESP-IDF Hooks do not operate in the same way as Vanilla FreeRTOS Hooks where users provide a definition for each of the hooks. Instead, the ESP-IDF Hooks are predefined to call a list of user registered callbacks specific to each core. Users can register and deregister callbacks which are run on the Idle or Tick Hook of a specific core.

### API Reference

#### Header File

- `esp32/include/esp_freertos_hooks.h`

## Functions

`esp_err_t esp_register_freertos_idle_hook_for_cpu(esp_freertos_idle_cb_t new_idle_cb, UBaseType_t cpuid)`

Register a callback to be called from the specified core's idle hook. The callback should return true if it should be called by the idle hook once per interrupt (or FreeRTOS tick), and return false if it should be called repeatedly as fast as possible by the idle hook.

**Warning** Idle callbacks MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

### Return

- ESP\_OK: Callback registered to the specified core's idle hook
- ESP\_ERR\_NO\_MEM: No more space on the specified core's idle hook to register callback
- ESP\_ERR\_INVALID\_ARG: cpuid is invalid

### Parameters

- `new_idle_cb`: Callback to be called
- `cpuid`: id of the core

`esp_err_t esp_register_freertos_idle_hook(esp_freertos_idle_cb_t new_idle_cb)`

Register a callback to the idle hook of the core that calls this function. The callback should return true if it should be called by the idle hook once per interrupt (or FreeRTOS tick), and return false if it should be called repeatedly as fast as possible by the idle hook.

**Warning** Idle callbacks MUST NOT, UNDER ANY CIRCUMSTANCES, CALL A FUNCTION THAT MIGHT BLOCK.

### Return

- ESP\_OK: Callback registered to the calling core's idle hook
- ESP\_ERR\_NO\_MEM: No more space on the calling core's idle hook to register callback

### Parameters

- `new_idle_cb`: Callback to be called

`esp_err_t esp_register_freertos_tick_hook_for_cpu(esp_freertos_tick_cb_t new_tick_cb, UBaseType_t cpuid)`

Register a callback to be called from the specified core's tick hook.

### Return

- ESP\_OK: Callback registered to specified core's tick hook
- ESP\_ERR\_NO\_MEM: No more space on the specified core's tick hook to register the callback
- ESP\_ERR\_INVALID\_ARG: cpuid is invalid

### Parameters

- `new_tick_cb`: Callback to be called
- `cpuid`: id of the core

`esp_err_t esp_register_freertos_tick_hook(esp_freertos_tick_cb_t new_tick_cb)`

Register a callback to be called from the calling core's tick hook.

**Return**

- ESP\_OK: Callback registered to the calling core's tick hook
- ESP\_ERR\_NO\_MEM: No more space on the calling core's tick hook to register the callback

**Parameters**

- `new_tick_cb`: Callback to be called

```
void esp_deregister_freertos_idle_hook_for_cpu (esp_freertos_idle_cb_t old_idle_cb,  
                                               UBaseType_t cpuid)
```

Unregister an idle callback from the idle hook of the specified core.

**Parameters**

- `old_idle_cb`: Callback to be unregistered
- `cpuid`: id of the core

```
void esp_deregister_freertos_idle_hook (esp_freertos_idle_cb_t old_idle_cb)
```

Unregister an idle callback. If the idle callback is registered to the idle hooks of both cores, the idle hook will be unregistered from both cores.

**Parameters**

- `old_idle_cb`: Callback to be unregistered

```
void esp_deregister_freertos_tick_hook_for_cpu (esp_freertos_tick_cb_t old_tick_cb,  
                                               UBaseType_t cpuid)
```

Unregister a tick callback from the tick hook of the specified core.

**Parameters**

- `old_tick_cb`: Callback to be unregistered
- `cpuid`: id of the core

```
void esp_deregister_freertos_tick_hook (esp_freertos_tick_cb_t old_tick_cb)
```

Unregister a tick callback. If the tick callback is registered to the tick hooks of both cores, the tick hook will be unregistered from both cores.

**Parameters**

- `old_tick_cb`: Callback to be unregistered

**Type Definitions**

```
typedef bool (*esp_freertos_idle_cb_t) ()
```

```
typedef void (*esp_freertos_tick_cb_t) ()
```

## 2.7.3 Heap Memory Allocation

## Overview

The ESP32 has multiple types of RAM. Internally, there's IRAM, DRAM as well as RAM that can be used as both. It's also possible to connect external SPI RAM to the ESP32 - external RAM can be integrated into the ESP32's memory map using the flash cache.

For most purposes, the standard `libc malloc()` and `free()` functions can be used for heap allocation without any issues.

However, in order to fully make use of all of the memory types and their characteristics, `esp-idf` also has a capabilities-based heap memory allocator. If you want to have memory with certain properties (for example, DMA-capable memory, or executable memory), you can create an OR-mask of the required capabilities and pass that to `heap_caps_malloc()`. For instance, the standard `malloc()` implementation internally allocates memory via `heap_caps_malloc(size, MALLOC_CAP_8BIT)` in order to get data memory that is byte-addressable.

Because `malloc` uses this allocation system as well, memory allocated using `heap_caps_malloc()` can be freed by calling the standard `free()` function.

The "soc" component contains a list of memory regions for the chip, along with the type of each memory (aka its tag) and the associated capabilities for that memory type. On startup, a separate heap is initialised for each contiguous memory region. The capabilities-based allocator chooses the best heap for each allocation, based on the requested capabilities.

## Special Uses

### DMA-Capable Memory

Use the `MALLOC_CAP_DMA` flag to allocate memory which is suitable for use with hardware DMA engines (for example SPI and I2S). This capability flag excludes any external PSRAM.

### 32-Bit Accessible Memory

If a certain memory structure is only addressed in 32-bit units, for example an array of ints or pointers, it can be useful to allocate it with the `MALLOC_CAP_32BIT` flag. This also allows the allocator to give out IRAM memory; something which it can't do for a normal `malloc()` call. This can help to use all the available memory in the ESP32.

Memory allocated with `MALLOC_CAP_32BIT` can *only* be accessed via 32-bit reads and writes, any other type of access will generate a fatal `LoadStoreError` exception.

## API Reference - Heap Allocation

### Header File

- `heap/include/esp_heap_caps.h`

### Functions

`void *heap_caps_malloc(size_t size, uint32_t caps)`

Allocate a chunk of memory which has the given capabilities.

Equivalent semantics to `libc malloc()`, for capability-aware memory.

In IDF, `malloc(p)` is equivalent to `heap_caps_malloc(p, MALLOC_CAP_8BIT)`.

**Return** A pointer to the memory allocated on success, NULL on failure

**Parameters**

- `size`: Size, in bytes, of the amount of memory to allocate
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory to be returned

void **heap\_caps\_free** (void \**ptr*)

Free memory previously allocated via `heap_caps_malloc()` or `heap_caps_realloc()`.

Equivalent semantics to `libc free()`, for capability-aware memory.

In IDF, `free (p)` is equivalent to `heap_caps_free (p)`.

**Parameters**

- `ptr`: Pointer to memory previously returned from `heap_caps_malloc()` or `heap_caps_realloc()`. Can be NULL.

void \***heap\_caps\_realloc** (void \**ptr*, size\_t *size*, int *caps*)

Reallocate memory previously allocated via `heap_caps_malloc()` or `heap_caps_realloc()`.

Equivalent semantics to `libc realloc()`, for capability-aware memory.

In IDF, `realloc (p, s)` is equivalent to `heap_caps_realloc (p, s, MALLOC_CAP_8BIT)`.

'caps' parameter can be different to the capabilities that any original 'ptr' was allocated with. In this way, `realloc` can be used to "move" a buffer if necessary to ensure it meets a new set of capabilities.

**Return** Pointer to a new buffer of size 'size' with capabilities 'caps', or NULL if allocation failed.

**Parameters**

- `ptr`: Pointer to previously allocated memory, or NULL for a new allocation.
- `size`: Size of the new buffer requested, or 0 to free the buffer.
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory desired for the new allocation.

void \***heap\_caps\_malloc** (size\_t *n*, size\_t *size*, uint32\_t *caps*)

Allocate a chunk of memory which has the given capabilities. The initialized value in the memory is set to zero.

Equivalent semantics to `libc calloc()`, for capability-aware memory.

In IDF, `calloc (p)` is equivalent to `heap_caps_malloc (p, MALLOC_CAP_8BIT)`.

**Return** A pointer to the memory allocated on success, NULL on failure

**Parameters**

- `n`: Number of continuing chunks of memory to allocate
- `size`: Size, in bytes, of a chunk of memory to allocate
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory to be returned

size\_t **heap\_caps\_get\_free\_size** (uint32\_t *caps*)

Get the total free size of all the regions that have the given capabilities.

This function takes all regions capable of having the given capabilities allocated in them and adds up the free space they have.

Note that because of heap fragmentation it is probably not possible to allocate a single block of memory of this size. Use `heap_caps_get_largest_free_block()` for this purpose.

**Return** Amount of free bytes in the regions

**Parameters**

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`size_t heap_caps_get_minimum_free_size (uint32_t caps)`

Get the total minimum free memory of all regions with the given capabilities.

This adds all the low water marks of the regions capable of delivering the memory with the given capabilities.

Note the result may be less than the global all-time minimum available heap of this kind, as “low water marks” are tracked per-region. Individual regions’ heaps may have reached their “low water marks” at different points in time. However this result still gives a “worst case” indication for all-time minimum free heap.

**Return** Amount of free bytes in the regions

**Parameters**

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`size_t heap_caps_get_largest_free_block (uint32_t caps)`

Get the largest free block of memory able to be allocated with the given capabilities.

Returns the largest value of `s` for which `heap_caps_malloc(s, caps)` will succeed.

**Return** Size of largest free block in bytes.

**Parameters**

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`void heap_caps_get_info (multi_heap_info_t *info, uint32_t caps)`

Get heap info for all regions with the given capabilities.

Calls `multi_heap_info()` on all heaps which share the given capabilities. The information returned is an aggregate across all matching heaps. The meanings of fields are the same as defined for `multi_heap_info_t`, except that `minimum_free_bytes` has the same caveats described in `heap_caps_get_minimum_free_size()`.

**Parameters**

- `info`: Pointer to a structure which will be filled with relevant heap metadata.
- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

`void heap_caps_print_heap_info (uint32_t caps)`

Print a summary of all memory with the given capabilities.

Calls `multi_heap_info` on all heaps which share the given capabilities, and prints a two-line summary for each, then a total summary.

**Parameters**

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

bool **heap\_caps\_check\_integrity\_all** (bool *print\_errors*)

Check integrity of all heap memory in the system.

Calls `multi_heap_check` on all heaps. Optionally print errors if heaps are corrupt.

Calling this function is equivalent to calling `heap_caps_check_integrity` with the `caps` argument set to `MALLOC_CAP_INVALID`.

**Return** True if all heaps are valid, False if at least one heap is corrupt.

#### Parameters

- `print_errors`: Print specific errors if heap corruption is found.

bool **heap\_caps\_check\_integrity** (uint32\_t *caps*, bool *print\_errors*)

Check integrity of all heaps with the given capabilities.

Calls `multi_heap_check` on all heaps which share the given capabilities. Optionally print errors if the heaps are corrupt.

See also `heap_caps_check_integrity_all` to check all heap memory in the system and `heap_caps_check_integrity_addr` to check memory around a single address.

**Return** True if all heaps are valid, False if at least one heap is corrupt.

#### Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory
- `print_errors`: Print specific errors if heap corruption is found.

bool **heap\_caps\_check\_integrity\_addr** (intptr\_t *addr*, bool *print\_errors*)

Check integrity of heap memory around a given address.

This function can be used to check the integrity of a single region of heap memory, which contains the given address.

This can be useful if debugging heap integrity for corruption at a known address, as it has a lower overhead than checking all heap regions. Note that if the corrupt address moves around between runs (due to timing or other factors) then this approach won't work and you should call `heap_caps_check_integrity` or `heap_caps_check_integrity_all` instead.

**Note** The entire heap region around the address is checked, not only the adjacent heap blocks.

**Return** True if the heap containing the specified address is valid, False if at least one heap is corrupt or the address doesn't belong to a heap region.

#### Parameters

- `addr`: Address in memory. Check for corruption in region containing this address.
- `print_errors`: Print specific errors if heap corruption is found.

void **heap\_caps\_malloc\_extmem\_enable** (size\_t *limit*)

Enable `malloc()` in external memory and set limit below which `malloc()` attempts are placed in internal memory.

When external memory is in use, the allocation strategy is to initially try to satisfy smaller allocation requests with internal memory and larger requests with external memory. This sets the limit between the two, as well as generally enabling allocation in external memory.

#### Parameters

- `limit`: Limit, in bytes.

void **\*heap\_caps\_malloc\_prefer** (size\_t *size*, size\_t *num*, ...)  
Allocate a chunk of memory as preference in decreasing order.

**Attention** The variable parameters are bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory. This API prefers to allocate memory with the first parameter. If failed, allocate memory with the next parameter. It will try in this order until allocating a chunk of memory successfully or fail to allocate memories with any of the parameters.

**Return** A pointer to the memory allocated on success, NULL on failure

#### Parameters

- `size`: Size, in bytes, of the amount of memory to allocate
- `num`: Number of variable parameters

void **\*heap\_caps\_realloc\_prefer** (void \**ptr*, size\_t *size*, size\_t *num*, ...)  
Allocate a chunk of memory as preference in decreasing order.

**Return** Pointer to a new buffer of size 'size', or NULL if allocation failed.

#### Parameters

- `ptr`: Pointer to previously allocated memory, or NULL for a new allocation.
- `size`: Size of the new buffer requested, or 0 to free the buffer.
- `num`: Number of variable parameters

void **\*heap\_caps\_calloc\_prefer** (size\_t *n*, size\_t *size*, size\_t *num*, ...)  
Allocate a chunk of memory as preference in decreasing order.

**Return** A pointer to the memory allocated on success, NULL on failure

#### Parameters

- `n`: Number of continuing chunks of memory to allocate
- `size`: Size, in bytes, of a chunk of memory to allocate
- `num`: Number of variable parameters

void **heap\_caps\_dump** (uint32\_t *caps*)  
Dump the full structure of all heaps with matching capabilities.

Prints a large amount of output to serial (because of locking limitations, the output bypasses stdout/stderr). For each (variable sized) block in each matching heap, the following output is printed on a single line:

- Block address (the data buffer returned by malloc is 4 bytes after this if heap debugging is set to Basic, or 8 bytes otherwise).
- Data size (the data size may be larger than the size requested by malloc, either due to heap fragmentation or because of heap debugging level).
- Address of next block in the heap.
- If the block is free, the address of the next free block is also printed.

#### Parameters

- `caps`: Bitwise OR of `MALLOC_CAP_*` flags indicating the type of memory

void `heap_caps_dump_all` ()

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`.

## Macros

### **MALLOC\_CAP\_EXEC**

Flags to indicate the capabilities of the various memory systems.

Memory must be able to run executable code

### **MALLOC\_CAP\_32BIT**

Memory must allow for aligned 32-bit data accesses.

### **MALLOC\_CAP\_8BIT**

Memory must allow for 8/16/...-bit data accesses.

### **MALLOC\_CAP\_DMA**

Memory must be able to accessed by DMA.

### **MALLOC\_CAP\_PID2**

Memory must be mapped to PID2 memory space (PIDs are not currently used)

### **MALLOC\_CAP\_PID3**

Memory must be mapped to PID3 memory space (PIDs are not currently used)

### **MALLOC\_CAP\_PID4**

Memory must be mapped to PID4 memory space (PIDs are not currently used)

### **MALLOC\_CAP\_PID5**

Memory must be mapped to PID5 memory space (PIDs are not currently used)

### **MALLOC\_CAP\_PID6**

Memory must be mapped to PID6 memory space (PIDs are not currently used)

### **MALLOC\_CAP\_PID7**

Memory must be mapped to PID7 memory space (PIDs are not currently used)

### **MALLOC\_CAP\_SPIRAM**

Memory must be in SPI RAM.

### **MALLOC\_CAP\_INTERNAL**

Memory must be internal; specifically it should not disappear when flash/spiram cache is switched off.

### **MALLOC\_CAP\_DEFAULT**

Memory can be returned in a non-capability-specific memory allocation (e.g. `malloc()`, `calloc()`) call.

### **MALLOC\_CAP\_INVALID**

Memory can't be used / list end marker.

## Heap Tracing & Debugging

The following features are documented on the [Heap Memory Debugging](#) page:

- [Heap Information](#) (free space, etc.)

- *Heap Corruption Detection*
- *Heap Tracing* (memory leak detection, monitoring, etc.)

## API Reference - Initialisation

### Header File

- `heap/include/esp_heap_caps_init.h`

### Functions

void **heap\_caps\_init** ()

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** ()

Enable heap(s) in memory regions where the startup stacks are located.

On startup, the pro/app CPUs have a certain memory region they use as stack, so we cannot do allocations in the regions these stack frames are. When FreeRTOS is completely started, they do not use that memory anymore and heap(s) there can be enabled.

esp\_err\_t **heap\_caps\_add\_region** (intptr\_t *start*, intptr\_t *end*)

Add a region of memory to the collection of heaps at runtime.

Most memory regions are defined in `soc_memory_layout.c` for the SoC, and are registered via `heap_caps_init()`. Some regions can't be used immediately and are later enabled via `heap_caps_enable_nonos_stack_heaps()`.

Call this function to add a region of memory to the heap at some later time.

This function does not consider any of the "reserved" regions or other data in `soc_memory_layout`, caller needs to consider this themselves.

All memory within the region specified by `start` & `end` parameters must be otherwise unused.

The capabilities of the newly registered memory will be determined by the start address, as looked up in the regions specified in `soc_memory_layout.c`.

Use `heap_caps_add_region_with_caps()` to register a region with custom capabilities.

**Return** ESP\_OK on success, ESP\_ERR\_INVALID\_ARG if a parameter is invalid, ESP\_ERR\_NOT\_FOUND if the specified start address doesn't reside in a known region, or any error returned by `heap_caps_add_region_with_caps()`.

#### Parameters

- `start`: Start address of new region.
- `end`: End address of new region.

esp\_err\_t **heap\_caps\_add\_region\_with\_caps** (const uint32\_t *caps*[], intptr\_t *start*, intptr\_t *end*)

Add a region of memory to the collection of heaps at runtime, with custom capabilities.

Similar to `heap_caps_add_region()`, only custom memory capabilities are specified by the caller.

#### Return

- ESP\_OK on success

- `ESP_ERR_INVALID_ARG` if a parameter is invalid
- `ESP_ERR_NO_MEM` if no memory to register new heap.
- `ESP_FAIL` if region overlaps the start and/or end of an existing region

#### Parameters

- `caps`: Ordered array of capability masks for the new region, in order of priority. Must have length `SOC_MEMORY_TYPE_NO_PRIOS`. Does not need to remain valid after the call returns.
- `start`: Start address of new region.
- `end`: End address of new region.

## Implementation Notes

Knowledge about the regions of memory in the chip comes from the “soc” component, which contains memory layout information for the chip.

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. When you call a function in the heap capabilities API, it will find the most appropriate heap for the allocation (based on desired capabilities, available space, and preferences for each region’s use) and then call the `multi_heap` function to use the heap situation in that particular region.

## API Reference - Multi Heap API

(Note: The multi heap API is used internally by the heap capabilities allocator. Most IDF programs will never need to call this API directly.)

### Header File

- `heap/include/multi_heap.h`

### Functions

void **multi\_heap\_malloc** (*multi\_heap\_handle\_t* heap, size\_t size)

malloc() a buffer in a given heap

Semantics are the same as standard malloc(), only the returned buffer will be allocated in the specified heap.

**Return** Pointer to new memory, or NULL if allocation fails.

#### Parameters

- `heap`: Handle to a registered heap.
- `size`: Size of desired buffer.

void **multi\_heap\_free** (*multi\_heap\_handle\_t* heap, void \*p)

free() a buffer in a given heap.

Semantics are the same as standard free(), only the argument ‘p’ must be NULL or have been allocated in the specified heap.

**Parameters**

- `heap`: Handle to a registered heap.
- `p`: NULL, or a pointer previously returned from `multi_heap_malloc()` or `multi_heap_realloc()` for the same heap.

void **multi\_heap\_realloc** (*multi\_heap\_handle\_t heap*, void *\*p*, size\_t *size*)  
 realloc() a buffer in a given heap.

Semantics are the same as standard `realloc()`, only the argument ‘`p`’ must be NULL or have been allocated in the specified heap.

**Return** New buffer of ‘`size`’ containing contents of ‘`p`’, or NULL if reallocation failed.

**Parameters**

- `heap`: Handle to a registered heap.
- `p`: NULL, or a pointer previously returned from `multi_heap_malloc()` or `multi_heap_realloc()` for the same heap.
- `size`: Desired new size for buffer.

size\_t **multi\_heap\_get\_allocated\_size** (*multi\_heap\_handle\_t heap*, void *\*p*)  
 Return the size that a particular pointer was allocated with.

**Return** Size of the memory allocated at this block. May be more than the original size argument, due to padding and minimum block sizes.

**Parameters**

- `heap`: Handle to a registered heap.
- `p`: Pointer, must have been previously returned from `multi_heap_malloc()` or `multi_heap_realloc()` for the same heap.

*multi\_heap\_handle\_t* **multi\_heap\_register** (void *\*start*, size\_t *size*)  
 Register a new heap for use.

This function initialises a heap at the specified address, and returns a handle for future heap operations.

There is no equivalent function for deregistering a heap - if all blocks in the heap are free, you can immediately start using the memory for other purposes.

**Return** Handle of a new heap ready for use, or NULL if the heap region was too small to be initialised.

**Parameters**

- `start`: Start address of the memory to use for a new heap.
- `size`: Size (in bytes) of the new heap.

void **multi\_heap\_set\_lock** (*multi\_heap\_handle\_t heap*, void *\*lock*)  
 Associate a private lock pointer with a heap.

The lock argument is supplied to the `MULTI_HEAP_LOCK()` and `MULTI_HEAP_UNLOCK()` macros, defined in `multi_heap_platform.h`.

The lock in question must be recursive.

When the heap is first registered, the associated lock is NULL.

### Parameters

- `heap`: Handle to a registered heap.
- `lock`: Optional pointer to a locking structure to associate with this heap.

void **multi\_heap\_dump** (*multi\_heap\_handle\_t heap*)

Dump heap information to stdout.

For debugging purposes, this function dumps information about every block in the heap to stdout.

### Parameters

- `heap`: Handle to a registered heap.

bool **multi\_heap\_check** (*multi\_heap\_handle\_t heap*, bool *print\_errors*)

Check heap integrity.

Walks the heap and checks all heap data structures are valid. If any errors are detected, an error-specific message can be optionally printed to stderr. Print behaviour can be overridden at compile time by defining `MULTI_CHECK_FAIL_PRINTF` in `multi_heap_platform.h`.

**Return** true if heap is valid, false otherwise.

### Parameters

- `heap`: Handle to a registered heap.
- `print_errors`: If true, errors will be printed to stderr.

size\_t **multi\_heap\_free\_size** (*multi\_heap\_handle\_t heap*)

Return free heap size.

Returns the number of bytes available in the heap.

Equivalent to the `total_free_bytes` member returned by `multi_heap_get_heap_info()`.

Note that the heap may be fragmented, so the actual maximum size for a single `malloc()` may be lower. To know this size, see the `largest_free_block` member returned by `multi_heap_get_heap_info()`.

**Return** Number of free bytes.

### Parameters

- `heap`: Handle to a registered heap.

size\_t **multi\_heap\_minimum\_free\_size** (*multi\_heap\_handle\_t heap*)

Return the lifetime minimum free heap size.

Equivalent to the `minimum_free_bytes` member returned by `multi_heap_get_info()`.

Returns the lifetime “low water mark” of possible values returned from `multi_free_heap_size()`, for the specified heap.

**Return** Number of free bytes.

### Parameters

- `heap`: Handle to a registered heap.

void **multi\_heap\_get\_info** (*multi\_heap\_handle\_t* heap, *multi\_heap\_info\_t* \*info)

Return metadata about a given heap.

Fills a *multi\_heap\_info\_t* structure with information about the specified heap.

#### Parameters

- heap: Handle to a registered heap.
- info: Pointer to a structure to fill with heap metadata.

### Structures

**struct multi\_heap\_info\_t**

Structure to access heap metadata via `multi_heap_get_info`.

#### Public Members

size\_t **total\_free\_bytes**

Total free bytes in the heap. Equivalent to `multi_free_heap_size()`.

size\_t **total\_allocated\_bytes**

Total bytes allocated to data in the heap.

size\_t **largest\_free\_block**

Size of largest free block in the heap. This is the largest malloc-able size.

size\_t **minimum\_free\_bytes**

Lifetime minimum free heap size. Equivalent to `multi_minimum_free_heap_size()`.

size\_t **allocated\_blocks**

Number of (variable size) blocks allocated in the heap.

size\_t **free\_blocks**

Number of (variable size) free blocks in the heap.

size\_t **total\_blocks**

Total number of (variable size) blocks in the heap.

### Type Definitions

**typedef struct** multi\_heap\_info \***multi\_heap\_handle\_t**

Opaque handle to a registered heap.

## 2.7.4 Heap Memory Debugging

### Overview

ESP-IDF integrates tools for requesting *heap information*, detecting heap corruption, and tracing memory leaks. These can help track down memory-related bugs.

For general information about the heap memory allocator, see the *Heap Memory Allocation* page.

## Heap Information

To obtain information about the state of the heap:

- `xPortGetFreeHeapSize()` is a FreeRTOS function which returns the number of free bytes in the (data memory) heap. This is equivalent to calling `heap_caps_get_free_size(MALLOC_CAP_8BIT)`.
- `heap_caps_get_free_size()` can also be used to return the current free memory for different memory capabilities.
- `heap_caps_get_largest_free_block()` can be used to return the largest free block in the heap. This is the largest single allocation which is currently possible. Tracking this value and comparing to total free heap allows you to detect heap fragmentation.
- `xPortGetMinimumEverFreeHeapSize()` and the related `heap_caps_get_minimum_free_size()` can be used to track the heap “low water mark” since boot.
- `heap_caps_get_info()` returns a `multi_heap_info_t` structure which contains the information from the above functions, plus some additional heap-specific data (number of allocations, etc.).
- `heap_caps_print_heap_info()` prints a summary to stdout of the information returned by `heap_caps_get_info()`.
- `heap_caps_dump()` and `heap_caps_dump_all()` will output detailed information about the structure of each block in the heap. Note that this can be large amount of output.

## Heap Corruption Detection

Heap corruption detection allows you to detect various types of heap memory errors:

- Out of bounds writes & buffer overflow.
- Writes to freed memory.
- Reads from freed or uninitialized memory,

## Assertions

The heap implementation (`multi_heap.c`, etc.) includes a lot of assertions which will fail if the heap memory is corrupted. To detect heap corruption most effectively, ensure that assertions are enabled in `make menuconfig` under `Compiler options`.

If a heap integrity assertion fails, a line will be printed like `CORRUPT HEAP: multi_heap.c:225 detected at 0x3ffbb71c`. The memory address which is printed is the address of the heap structure which has corrupt content.

It’s also possible to manually check heap integrity by calling `heap_caps_check_integrity_all()` or related functions. This function checks all of requested heap memory for integrity, and can be used even if assertions are disabled. If the integrity check prints an error, it will also contain the address(es) of corrupt heap structures.

## Finding Heap Corruption

Memory corruption can be one of the hardest classes of bugs to find and fix, as one area of memory can be corrupted from a totally different place. Some tips:

- A crash with a `CORRUPT HEAP :` message will usually include a stack trace, but this stack trace is rarely useful. The crash is the symptom of memory corruption when the system realises the heap is corrupt, but usually the corruption happened elsewhere and earlier in time.

- Increasing the Heap memory debugging *Configuration* level to “Light impact” or “Comprehensive” can give you a more accurate message with the first corrupt memory address.
- Adding regular calls to `heap_caps_check_integrity_all()` or `heap_caps_check_integrity_addr()` in your code will help you pin down the exact time that the corruption happened. You can move these checks around to “close in on” the section of code that corrupted the heap.
- Based on the memory address which is being corrupted, you can use *JTAG debugging* to set a watchpoint on this address and have the CPU halt when it is written to.
- If you don’t have JTAG, but you do know roughly when the corruption happens, then you can set a watchpoint in software just beforehand via `esp_set_watchpoint()`. A fatal exception will occur when the watchpoint triggers. For example `esp_set_watchpoint(0, (void *)addr, 4, ESP_WATCHPOINT_STORE)`. Note that watchpoints are per-CPU and are set on the current running CPU only, so if you don’t know which CPU is corrupting memory then you will need to call this function on both CPUs.
- For buffer overflows, *heap tracing* in `HEAP_TRACE_ALL` mode lets you see which callers are allocating which addresses from the heap. See *Heap Tracing To Find Heap Corruption* for more details. If you can find the function which allocates memory with an address immediately before the address which is corrupted, this will probably be the function which overflows the buffer.
- Calling `heap_caps_dump()` or `heap_caps_dump_all()` can give an indication of what heap blocks are surrounding the corrupted region and may have overflowed/underflowed/etc.

## Configuration

Temporarily increasing the heap corruption detection level can give more detailed information about heap corruption errors.

In `make menuconfig`, under `Component config` there is a menu `Heap memory debugging`. The setting `HEAP_CORRUPTION_DETECTION` can be set to one of three levels:

### Basic (no poisoning)

This is the default level. No special heap corruption features are enabled, but provided assertions are enabled (the default configuration) then a heap corruption error will be printed if any of the heap’s internal data structures appear overwritten or corrupted. This usually indicates a buffer overrun or out of bounds write.

If assertions are enabled, an assertion will also trigger if a double-free occurs (the same memory is freed twice).

Calling `heap_caps_check_integrity()` in Basic mode will check the integrity of all heap structures, and print errors if any appear to be corrupted.

### Light Impact

At this level, heap memory is additionally “poisoned” with head and tail “canary bytes” before and after each block which is allocated. If an application writes outside the bounds of allocated buffers, the canary bytes will be corrupted and the integrity check will fail.

The head canary word is `0xABBA1234` (`3412BAAB` in byte order), and the tail canary word is `0xBAAD5678` (`7856ADBA` in byte order).

“Basic” heap corruption checks can also detect most out of bounds writes, but this setting is more precise as even a single byte overrun can be detected. With Basic heap checks, the number of overrun bytes before a failure is detected will depend on the properties of the heap.

Enabling “Light Impact” checking increases memory usage, each individual allocation will use 9 to 12 additional bytes of memory (depending on alignment).

Each time `free()` is called in Light Impact mode, the head and tail canary bytes of the buffer being freed are checked against the expected values.

When `heap_caps_check_integrity()` is called, all allocated blocks of heap memory have their canary bytes checked against the expected values.

In both cases, the check is that the first 4 bytes of an allocated block (before the buffer returned to the user) should be the word 0xABBA1234. Then the last 4 bytes of the allocated block (after the buffer returned to the user) should be the word 0xBAAD5678.

Different values usually indicate buffer underrun or overrun, respectively.

### Comprehensive

This level incorporates the “light impact” detection features plus additional checks for uninitialised-access and use-after-free bugs. In this mode, all freshly allocated memory is filled with the pattern 0xCE, and all freed memory is filled with the pattern 0xFE.

Enabling “Comprehensive” detection has a substantial runtime performance impact (as all memory needs to be set to the allocation patterns each time a malloc/free completes, and the memory also needs to be checked each time.) However it allows easier detection of memory corruption bugs which are much more subtle to find otherwise. It is recommended to only enable this mode when debugging, not in production.

### Crashes in Comprehensive Mode

If an application crashes reading/writing an address related to 0xCECECECE in Comprehensive mode, this indicates it has read uninitialized memory. The application should be changed to either use `calloc()` (which zeroes memory), or initialize the memory before using it. The value 0xCECECECE may also be seen in stack-allocated automatic variables, because in IDF most task stacks are originally allocated from the heap and in C stack memory is uninitialized by default.

If an application crashes and the exception register dump indicates that some addresses or values were 0xFEFEFEFE, this indicates it is reading heap memory after it has been freed (a “use after free bug”.) The application should be changed to not access heap memory after it has been freed.

If a call to `malloc()` or `realloc()` causes a crash because it expected to find the pattern 0xFEFEFEFE in free memory and a different pattern was found, then this indicates the app has a use-after-free bug where it is writing to memory which has already been freed.

### Manual Heap Checks in Comprehensive Mode

Calls to `heap_caps_check_integrity()` may print errors relating to 0xFEFEFEFE, 0xABBA1234 or 0xBAAD5678. In each case the checker is expecting to find a given pattern, and will error out if this is not found:

- For free heap blocks, the checker expects to find all bytes set to 0xFE. Any other values indicate a use-after-free bug where free memory has been incorrectly overwritten.
- For allocated heap blocks, the behaviour is the same as for *Light Impact* mode. The canary bytes 0xABBA1234 and 0xBAAD5678 are checked at the head and tail of each allocated buffer, and any variation indicates a buffer overrun/underrun.

## Heap Tracing

Heap Tracing allows tracing of code which allocates/frees memory.

---

**Note:** Heap tracing “standalone” mode is currently implemented, meaning that tracing does not require any external hardware but uses internal memory to hold trace data. Heap tracing via JTAG trace port is also planned.

---

Heap tracing can perform two functions:

- Leak checking: find memory which is allocated and never freed.
- Heap use analysis: show all functions that are allocating/freeing memory while the trace is running.

## How To Diagnose Memory Leaks

If you suspect a memory leak, the first step is to figure out which part of the program is leaking memory. Use the `xPortGetFreeHeapSize()`, `heap_caps_get_free()`, and related functions to track memory use over the life of the application. Try to narrow the leak down to a single function or sequence of functions where free memory always decreases and never recovers.

Once you’ve identified the code which you think is leaking:

- Under `make menuconfig`, navigate to `Component settings -> Heap Memory Debugging` and set `HEAP_TRACING`.
- Call the function `heap_trace_init_standalone()` early in the program, to register a buffer which can be used to record the memory trace.
- Call the function `heap_trace_start()` to begin recording all mallocs/frees in the system. Call this immediately before the piece of code which you suspect is leaking memory.
- Call the function `heap_trace_stop()` to stop the trace once the suspect piece of code has finished executing.
- Call the function `heap_trace_dump()` to dump the results of the heap trace.

An example:

```
#include "esp_heap_trace.h"

#define NUM_RECORDS 100
static heap_trace_record_t trace_record[NUM_RECORDS]; // This buffer must be in_
↳ internal RAM

...

void app_main()
{
    ...
    ESP_ERROR_CHECK( heap_trace_init_standalone(trace_record, NUM_RECORDS) );
    ...
}

void some_function()
{
    ESP_ERROR_CHECK( heap_trace_start(HEAP_TRACE_LEAKS) );
```

(continues on next page)

(continued from previous page)

```

do_something_you_suspect_is_leaking();

ESP_ERROR_CHECK( heap_trace_stop() );
heap_trace_dump();
...
}

```

The output from the heap trace will look something like this:

```

2 allocations trace (100 entry buffer)
32 bytes (@ 0x3ffaf214) allocated CPU 0 ccount 0x2e9b7384 caller 0x400d276d:0x400d27c1
0x400d276d: leak_some_memory at /path/to/idf/examples/get-started/blink/main/./blink.
↪c:27

0x400d27c1: blink_task at /path/to/idf/examples/get-started/blink/main/./blink.c:52

8 bytes (@ 0x3ffaf804) allocated CPU 0 ccount 0x2e9b79c0 caller 0x400d2776:0x400d27c1
0x400d2776: leak_some_memory at /path/to/idf/examples/get-started/blink/main/./blink.
↪c:29

0x400d27c1: blink_task at /path/to/idf/examples/get-started/blink/main/./blink.c:52

40 bytes 'leaked' in trace (2 allocations)
total allocations 2 total frees 0

```

(Above example output is using *IDF Monitor* to automatically decode PC addresses to their source files & line number.)

The first line indicates how many allocation entries are in the buffer, compared to its total size.

In `HEAP_TRACE_LEAKS` mode, for each traced memory allocation which has not already been freed a line is printed with:

- `XX bytes` is number of bytes allocated
- `@ 0x...` is the heap address returned from `malloc/calloc`.
- `CPU x` is the CPU (0 or 1) running when the allocation was made.
- `ccount 0x...` is the `CCOUNT` (CPU cycle count) register value when the allocation was made. Is different for CPU 0 vs CPU 1.
- `caller 0x...` gives the call stack of the call to `malloc()/free()`, as a list of PC addresses. These can be decoded to source files and line numbers, as shown above.

The depth of the call stack recorded for each trace entry can be configured in `make menuconfig`, under `Heap Memory Debugging` -> `Enable heap tracing` -> `Heap tracing stack depth`. Up to 10 stack frames can be recorded for each allocation (the default is 2). Each additional stack frame increases the memory usage of each `heap_trace_record_t` record by eight bytes.

Finally, the total number of 'leaked' bytes (bytes allocated but not freed while trace was running) is printed, and the total number of allocations this represents.

A warning will be printed if the trace buffer was not large enough to hold all the allocations which happened. If you see this warning, consider either shortening the tracing period or increasing the number of records in the trace buffer.

## Heap Tracing To Find Heap Corruption

When a region in heap is corrupted, it may be from some other part of the program which allocated memory at a nearby address.

If you have some idea at what time the corruption occurred, enabling heap tracing in `HEAP_TRACE_ALL` mode allows you to record all of the functions which allocated memory, and the addresses where they were corrupted.

Using heap tracing in this way is very similar to memory leak detection as described above. For memory which is allocated and not freed, the output

Heap tracing can also be used to help track down heap corruption. By using

## Performance Impact

Enabling heap tracing in menuconfig increases the code size of your program, and has a very small negative impact on performance of heap allocation/free operations even when heap tracing is not running.

When heap tracing is running, heap allocation/free operations are substantially slower than when heap tracing is stopped. Increasing the depth of stack frames recorded for each allocation (see above) will also increase this performance impact.

## False-Positive Memory Leaks

Not everything printed by `heap_trace_dump()` is necessarily a memory leak. Among things which may show up here, but are not memory leaks:

- Any memory which is allocated after `heap_trace_start()` but then freed after `heap_trace_stop()` will appear in the leak dump.
- Allocations may be made by other tasks in the system. Depending on the timing of these tasks, it's quite possible this memory is freed after `heap_trace_stop()` is called.
- The first time a task uses stdio - for example, when it calls `printf()` - a lock (RTOS mutex semaphore) is allocated by the libc. This allocation lasts until the task is deleted.
- The Bluetooth, WiFi, and TCP/IP libraries will allocate heap memory buffers to handle incoming or outgoing data. These memory buffers are usually short lived, but some may be shown in the heap leak trace if the data was received/transmitted by the lower levels of the network while the leak trace was running.
- TCP connections will continue to use some memory after they are closed, because of the `TIME_WAIT` state. After the `TIME_WAIT` period has completed, this memory will be freed.

One way to differentiate between “real” and “false positive” memory leaks is to call the suspect code multiple times while tracing is running, and look for patterns (multiple matching allocations) in the heap trace output.

## API Reference - Heap Tracing

### Header File

- `heap/include/esp_heap_trace.h`

## Functions

`esp_err_t heap_trace_init_standalone(heap_trace_record_t *record_buffer, size_t num_records)`

Initialise heap tracing in standalone mode.

This function must be called before any other heap tracing functions.

**Note** Standalone mode is the only mode currently supported.

To disable heap tracing and allow the buffer to be freed, stop tracing and then call `heap_trace_init_standalone(NULL, 0)`;

### Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` Heap tracing is currently in progress.
- `ESP_OK` Heap tracing initialised successfully.

### Parameters

- `record_buffer`: Provide a buffer to use for heap trace data. Must remain valid any time heap tracing is enabled, meaning it must be allocated from internal memory not in PSRAM.
- `num_records`: Size of the heap trace buffer, as number of record structures.

`esp_err_t heap_trace_start(heap_trace_mode_t mode)`

Start heap tracing. All heap allocations & frees will be traced, until `heap_trace_stop()` is called.

**Note** `heap_trace_init_standalone()` must be called to provide a valid buffer, before this function is called.

**Note** Calling this function while heap tracing is running will reset the heap trace state and continue tracing.

### Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` A non-zero-length buffer has not been set via `heap_trace_init_standalone()`.
- `ESP_OK` Tracing is started.

### Parameters

- `mode`: Mode for tracing.
  - `HEAP_TRACE_ALL` means all heap allocations and frees are traced.
  - `HEAP_TRACE_LEAKS` means only suspected memory leaks are traced. (When memory is freed, the record is removed from the trace buffer.)

`esp_err_t heap_trace_stop(void)`

Stop heap tracing.

### Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` Heap tracing was not in progress.
- `ESP_OK` Heap tracing stopped..

esp\_err\_t **heap\_trace\_resume** (void)

Resume heap tracing which was previously stopped.

Unlike `heap_trace_start()`, this function does not clear the buffer of any pre-existing trace records.

The heap trace mode is the same as when `heap_trace_start()` was last called (or `HEAP_TRACE_ALL` if `heap_trace_start()` was never called).

#### Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` Heap tracing was already started.
- `ESP_OK` Heap tracing resumed.

size\_t **heap\_trace\_get\_count** (void)

Return number of records in the heap trace buffer.

It is safe to call this function while heap tracing is running.

esp\_err\_t **heap\_trace\_get** (size\_t *index*, *heap\_trace\_record\_t* \**record*)

Return a raw record from the heap trace buffer.

**Note** It is safe to call this function while heap tracing is running, however in `HEAP_TRACE_LEAK` mode record indexing may skip entries unless heap tracing is stopped first.

#### Return

- `ESP_ERR_NOT_SUPPORTED` Project was compiled without heap tracing enabled in menuconfig.
- `ESP_ERR_INVALID_STATE` Heap tracing was not initialised.
- `ESP_ERR_INVALID_ARG` Index is out of bounds for current heap trace record count.
- `ESP_OK` Record returned successfully.

#### Parameters

- *index*: Index (zero-based) of the record to return.
- *record*: Record where the heap trace record will be copied.

void **heap\_trace\_dump** (void)

Dump heap trace record data to stdout.

**Note** It is safe to call this function while heap tracing is running, however in `HEAP_TRACE_LEAK` mode the dump may skip entries unless heap tracing is stopped first.

## Structures

**struct heap\_trace\_record\_t**

Trace record data type. Stores information about an allocated region of memory.

#### Public Members

uint32\_t **ccount**

CCOUNT of the CPU when the allocation was made. LSB (bit value 1) is the CPU number (0 or 1). \*/.

void **\*address**  
Address which was allocated.

size\_t **size**  
Size of the allocation.

void **\*allocated\_by**[CONFIG\_HEAP\_TRACING\_STACK\_DEPTH]  
Call stack of the caller which allocated the memory.

void **\*freed\_by**[CONFIG\_HEAP\_TRACING\_STACK\_DEPTH]  
Call stack of the caller which freed the memory (all zero if not freed.)

## Macros

**CONFIG\_HEAP\_TRACING\_STACK\_DEPTH**

## Enumerations

enum **heap\_trace\_mode\_t**  
*Values:*

**HEAP\_TRACE\_ALL**

**HEAP\_TRACE\_LEAKS**

## 2.7.5 Interrupt allocation

### Overview

The ESP32 has two cores, with 32 interrupts each. Each interrupt has a certain priority level, most (but not all) interrupts are connected to the interrupt mux. Because there are more interrupt sources than interrupts, sometimes it makes sense to share an interrupt in multiple drivers. The `esp_intr_alloc` abstraction exists to hide all these implementation details.

A driver can allocate an interrupt for a certain peripheral by calling `esp_intr_alloc` (or `esp_intr_alloc_sintrstatus`). It can use the flags passed to this function to set the type of interrupt allocated, specifying a specific level or trigger method. The interrupt allocation code will then find an applicable interrupt, use the interrupt mux to hook it up to the peripheral, and install the given interrupt handler and ISR to it.

This code has two different types of interrupts it handles differently: Shared interrupts and non-shared interrupts. The simplest of the two are non-shared interrupts: a separate interrupt is allocated per `esp_intr_alloc` call and this interrupt is solely used for the peripheral attached to it, with only one ISR that will get called. Shared interrupts can have multiple peripherals triggering it, with multiple ISRs being called when one of the peripherals attached signals an interrupt. Thus, ISRs that are intended for shared interrupts should check the interrupt status of the peripheral they service in order to see if any action is required.

Non-shared interrupts can be either level- or edge-triggered. Shared interrupts can only be level interrupts (because of the chance of missed interrupts when edge interrupts are used.) (The logic behind this: DevA and DevB share an int. DevB signals an int. Int line goes high. ISR handler calls code for DevA -> does nothing. ISR handler calls code for DevB, but while doing that, DevA signals an int. ISR DevB is done, clears int for DevB, exits interrupt code. Now an interrupt for DevA is still pending, but because the int line never went low (DevA kept it high even when the int for DevB was cleared) the interrupt is never serviced.)

## Multicore issues

Peripherals that can generate interrupts can be divided in two types:

- External peripherals, within the ESP32 but outside the Xtensa cores themselves. Most ESP32 peripherals are of this type.
- Internal peripherals, part of the Xtensa CPU cores themselves.

Interrupt handling differs slightly between these two types of peripherals.

## Internal peripheral interrupts

Each Xtensa CPU core has its own set of six internal peripherals:

- Three timer comparators
- A performance monitor
- Two software interrupts.

Internal interrupt sources are defined in `esp_intr_alloc.h` as `ETS_INTERNAL_*_INTR_SOURCE`.

These peripherals can only be configured from the core they are associated with. When generating an interrupt, the interrupt they generate is hard-wired to their associated core; it's not possible to have e.g. an internal timer comparator of one core generate an interrupt on another core. That is why these sources can only be managed using a task running on that specific core. Internal interrupt sources are still allocatable using `esp_intr_alloc` as normal, but they cannot be shared and will always have a fixed interrupt level (namely, the one associated in hardware with the peripheral).

## External Peripheral Interrupts

The remaining interrupt sources are from external peripherals. These are defined in `soc/soc.h` as `ETS_*_INTR_SOURCE`.

Non-internal interrupt slots in both CPU cores are wired to an interrupt multiplexer, which can be used to route any external interrupt source to any of these interrupt slots.

- Allocating an external interrupt will always allocate it on the core that does the allocation.
- Freeing an external interrupt must always happen on the same core it was allocated on.
- Disabling and enabling external interrupts from another core is allowed.
- Multiple external interrupt sources can share an interrupt slot by passing `ESP_INTR_FLAG_SHARED` as a flag to `esp_intr_alloc()`.

Care should be taken when calling `esp_intr_alloc()` from a task which is not pinned to a core. During task switching, these tasks can migrate between cores. Therefore it is impossible to tell which CPU the interrupt is allocated on, which makes it difficult to free the interrupt handle and may also cause debugging difficulties. It is advised to use `xTaskCreatePinnedToCore()` with a specific `CoreID` argument to create tasks that will allocate interrupts. In the case of internal interrupt sources, this is required.

## IRAM-Safe Interrupt Handlers

The `ESP_INTR_FLAG_IRAM` flag registers an interrupt handler that always runs from IRAM (and reads all its data from DRAM), and therefore does not need to be disabled during flash erase and write operations.

This is useful for interrupts which need a guaranteed minimum execution latency, as flash write and erase operations can be slow (erases can take tens or hundreds of milliseconds to complete).

It can also be useful to keep an interrupt handler in IRAM if it is called very frequently, to avoid flash cache misses.

Refer to the *SPI flash API documentation* for more details.

## Multiple Handlers Sharing A Source

Several handlers can be assigned to a same source, given that all handlers are allocated using the `ESP_INTR_FLAG_SHARED` flag. They'll be all allocated to the interrupt, which the source is attached to, and called sequentially when the source is active. The handlers can be disabled and freed individually. The source is attached to the interrupt (enabled), if one or more handlers are enabled, otherwise detached. A handler will never be called when disabled, while **its source may still be triggered** if any one of its handler enabled.

Sources attached to non-shared interrupt do not support this feature.

Though the framework support this feature, you have to use it *very carefully*. There usually exist 2 ways to stop a interrupt from being triggered: *disable the source* or *mask peripheral interrupt status*. IDF only handles the enabling and disabling of the source itself, leaving status and mask bits to be handled by users. **Status bits should always be masked before the handler responsible for it is disabled, or the status should be handled in other enabled interrupt properly**. You may leave some status bits unhandled if you just disable one of all the handlers without mask the status bits, which causes the interrupt being triggered infinitely, and finally a system crash.

## API Reference

### Header File

- `esp32/include/esp_intr_alloc.h`

### Functions

`esp_err_t esp_intr_mark_shared(int intno, int cpu, bool is_in_iram)`

Mark an interrupt as a shared interrupt.

This will mark a certain interrupt on the specified CPU as an interrupt that can be used to hook shared interrupt handlers to.

**Return** `ESP_ERR_INVALID_ARG` if `cpu` or `intno` is invalid `ESP_OK` otherwise

#### Parameters

- `intno`: The number of the interrupt (0-31)
- `cpu`: CPU on which the interrupt should be marked as shared (0 or 1)
- `is_in_iram`: Shared interrupt is for handlers that reside in IRAM and the int can be left enabled while the flash cache is disabled.

`esp_err_t esp_intr_reserve(int intno, int cpu)`

Reserve an interrupt to be used outside of this framework.

This will mark a certain interrupt on the specified CPU as reserved, not to be allocated for any reason.

**Return** `ESP_ERR_INVALID_ARG` if `cpu` or `intno` is invalid `ESP_OK` otherwise

#### Parameters

- `intno`: The number of the interrupt (0-31)

- `cpu`: CPU on which the interrupt should be marked as shared (0 or 1)

`esp_err_t esp_intr_alloc` (int *source*, int *flags*, *intr\_handler\_t* *handler*, void *\*arg*, *intr\_handle\_t* *\*ret\_handle*)

Allocate an interrupt with the given parameters.

This finds an interrupt that matches the restrictions as given in the flags parameter, maps the given interrupt source to it and hooks up the given interrupt handler (with optional argument) as well. If needed, it can return a handle for the interrupt as well.

The interrupt will always be allocated on the core that runs this function.

If `ESP_INTR_FLAG_IRAM` flag is used, and handler address is not in IRAM or `RTC_FAST_MEM`, then `ESP_ERR_INVALID_ARG` is returned.

**Return** `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_ERR_NOT_FOUND` No free interrupt found with the specified flags `ESP_OK` otherwise

#### Parameters

- `source`: The interrupt source. One of the `ETS_*_INTR_SOURCE` interrupt mux sources, as defined in `soc/soc.h`, or one of the internal `ETS_INTERNAL_*_INTR_SOURCE` sources as defined in this header.
- `flags`: An ORred mask of the `ESP_INTR_FLAG_*` defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is `ESP_INTR_FLAG_SHARED`, it will allocate a shared interrupt of level 1. Setting `ESP_INTR_FLAG_INTRDISABLED` will return from this function with the interrupt disabled.
- `handler`: The interrupt handler. Must be `NULL` when an interrupt of level >3 is requested, because these types of interrupts aren't C-callable.
- `arg`: Optional argument for passed to the interrupt handler
- `ret_handle`: Pointer to an `intr_handle_t` to store a handle that can later be used to request details or free the interrupt. Can be `NULL` if no handle is required.

`esp_err_t esp_intr_alloc_intrstatus` (int *source*, int *flags*, uint32\_t *intrstatusreg*, uint32\_t *intrstatusmask*, *intr\_handler\_t* *handler*, void *\*arg*, *intr\_handle\_t* *\*ret\_handle*)

Allocate an interrupt with the given parameters.

This essentially does the same as `esp_intr_alloc`, but allows specifying a register and mask combo. For shared interrupts, the handler is only called if a read from the specified register, ANDed with the mask, returns non-zero. By passing an interrupt status register address and a fitting mask, this can be used to accelerate interrupt handling in the case a shared interrupt is triggered; by checking the interrupt statuses first, the code can decide which ISRs can be skipped

**Return** `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_ERR_NOT_FOUND` No free interrupt found with the specified flags `ESP_OK` otherwise

#### Parameters

- `source`: The interrupt source. One of the `ETS_*_INTR_SOURCE` interrupt mux sources, as defined in `soc/soc.h`, or one of the internal `ETS_INTERNAL_*_INTR_SOURCE` sources as defined in this header.
- `flags`: An ORred mask of the `ESP_INTR_FLAG_*` defines. These restrict the choice of interrupts that this routine can choose from. If this value is 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is `ESP_INTR_FLAG_SHARED`, it will allocate a shared interrupt of level

1. Setting `ESP_INTR_FLAG_INTRDISABLED` will return from this function with the interrupt disabled.

- `intrstatusreg`: The address of an interrupt status register
- `intrstatusmask`: A mask. If a read of address `intrstatusreg` has any of the bits that are 1 in the mask set, the ISR will be called. If not, it will be skipped.
- `handler`: The interrupt handler. Must be `NULL` when an interrupt of level `>3` is requested, because these types of interrupts aren't C-callable.
- `arg`: Optional argument for passed to the interrupt handler
- `ret_handle`: Pointer to an `intr_handle_t` to store a handle that can later be used to request details or free the interrupt. Can be `NULL` if no handle is required.

`esp_err_t esp_intr_free(intr_handle_t handle)`

Disable and free an interrupt.

Use an interrupt handle to disable the interrupt and release the resources associated with it.

**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.

**Return** `ESP_ERR_INVALID_ARG` if handle is invalid, or `esp_intr_free` runs on another core than where the interrupt is allocated on. `ESP_OK` otherwise

#### Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`int esp_intr_get_cpu(intr_handle_t handle)`

Get CPU number an interrupt is tied to.

**Return** The core number where the interrupt is allocated

#### Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`int esp_intr_get_intno(intr_handle_t handle)`

Get the allocated interrupt for a certain handle.

**Return** The interrupt number

#### Parameters

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`esp_err_t esp_intr_disable(intr_handle_t handle)`

Disable the interrupt associated with the handle.

#### Note

1. For local interrupts (`ESP_INTERNAL_*` sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.
2. When several handlers sharing a same interrupt source, interrupt status bits, which are handled in the handler to be disabled, should be masked before the disabling, or handled in other enabled interrupts properly. Miss of interrupt status handling will cause infinite interrupt calls and finally system crash.

**Return** `ESP_ERR_INVALID_ARG` if the combination of arguments is invalid. `ESP_OK` otherwise

**Parameters**

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

`esp_err_t esp_intr_enable (intr_handle_t handle)`  
 Enable the interrupt associated with the handle.

**Note** For local interrupts (ESP\_INTERNAL\_\* sources), this function has to be called on the CPU the interrupt is allocated on. Other interrupts have no such restriction.

**Return** ESP\_ERR\_INVALID\_ARG if the combination of arguments is invalid. ESP\_OK otherwise

**Parameters**

- `handle`: The handle, as obtained by `esp_intr_alloc` or `esp_intr_alloc_intrstatus`

void `esp_intr_noniram_disable ()`  
 Disable interrupts that aren't specifically marked as running from IRAM.

void `esp_intr_noniram_enable ()`  
 Re-enable interrupts disabled by `esp_intr_noniram_disable`.

**Macros**

**ESP\_INTR\_FLAG\_LEVEL1**  
 Interrupt allocation flags.

These flags can be used to specify which interrupt qualities the code calling `esp_intr_alloc*` needs. Accept a Level 1 interrupt vector (lowest priority)

**ESP\_INTR\_FLAG\_LEVEL2**  
 Accept a Level 2 interrupt vector.

**ESP\_INTR\_FLAG\_LEVEL3**  
 Accept a Level 3 interrupt vector.

**ESP\_INTR\_FLAG\_LEVEL4**  
 Accept a Level 4 interrupt vector.

**ESP\_INTR\_FLAG\_LEVEL5**  
 Accept a Level 5 interrupt vector.

**ESP\_INTR\_FLAG\_LEVEL6**  
 Accept a Level 6 interrupt vector.

**ESP\_INTR\_FLAG\_NMI**  
 Accept a Level 7 interrupt vector (highest priority)

**ESP\_INTR\_FLAG\_SHARED**  
 Interrupt can be shared between ISRs.

**ESP\_INTR\_FLAG\_EDGE**  
 Edge-triggered interrupt.

**ESP\_INTR\_FLAG\_IRAM**  
 ISR can be called if cache is disabled.

**ESP\_INTR\_FLAG\_INTRDISABLED**  
 Return with this interrupt disabled.

**ESP\_INTR\_FLAG\_LOWMED**

Low and medium prio interrupts. These can be handled in C.

**ESP\_INTR\_FLAG\_HIGH**

High level interrupts. Need to be handled in assembly.

**ESP\_INTR\_FLAG\_LEVELMASK**

Mask for all level flags.

**ETS\_INTERNAL\_TIMER0\_INTR\_SOURCE**

Xtensa timer 0 interrupt source.

The `esp_intr_alloc*` functions can allocate an int for all `ETS_*_INTR_SOURCE` interrupt sources that are routed through the interrupt mux. Apart from these sources, each core also has some internal sources that do not pass through the interrupt mux. To allocate an interrupt for these sources, pass these pseudo-sources to the functions.

**ETS\_INTERNAL\_TIMER1\_INTR\_SOURCE**

Xtensa timer 1 interrupt source.

**ETS\_INTERNAL\_TIMER2\_INTR\_SOURCE**

Xtensa timer 2 interrupt source.

**ETS\_INTERNAL\_SW0\_INTR\_SOURCE**

Software int source 1.

**ETS\_INTERNAL\_SW1\_INTR\_SOURCE**

Software int source 2.

**ETS\_INTERNAL\_PROFILING\_INTR\_SOURCE**

Int source for profiling.

**ETS\_INTERNAL\_INTR\_SOURCE\_OFF**

## Type Definitions

```
typedef void (*intr_handler_t) (void *arg)
typedef struct intr_handle_data_t intr_handle_data_t
typedef intr_handle_data_t *intr_handle_t
```

## 2.7.6 Watchdogs

### Overview

The ESP-IDF has support for two types of watchdogs: The Interrupt Watchdog Timer and the Task Watchdog Timer (TWDT). The Interrupt Watchdog Timer and the TWDT can both be enabled using `make menuconfig`, however the TWDT can also be enabled during runtime. The Interrupt Watchdog is responsible for detecting instances where FreeRTOS task switching is blocked for a prolonged period of time. The TWDT is responsible for detecting instances of tasks running without yielding for a prolonged period.

### Interrupt watchdog

The interrupt watchdog makes sure the FreeRTOS task switching interrupt isn't blocked for a long time. This is bad because no other tasks, including potentially important ones like the WiFi task and the idle task, can't get any CPU

runtime. A blocked task switching interrupt can happen because a program runs into an infinite loop with interrupts disabled or hangs in an interrupt.

The default action of the interrupt watchdog is to invoke the panic handler, causing a register dump and an opportunity for the programmer to find out, using either OpenOCD or gdbstub, what bit of code is stuck with interrupts disabled. Depending on the configuration of the panic handler, it can also blindly reset the CPU, which may be preferred in a production environment.

The interrupt watchdog is built around the hardware watchdog in timer group 1. If this watchdog for some reason cannot execute the NMI handler that invokes the panic handler (e.g. because IRAM is overwritten by garbage), it will hard-reset the SOC.

## Task Watchdog Timer

The Task Watchdog Timer (TWDT) is responsible for detecting instances of tasks running for a prolonged period of time without yielding. This is a symptom of CPU starvation and is usually caused by a higher priority task looping without yielding to a lower-priority task thus starving the lower priority task from CPU time. This can be an indicator of poorly written code that spinloops on a peripheral, or a task that is stuck in an infinite loop.

By default the TWDT will watch the Idle Tasks of each CPU, however any task can elect to be watched by the TWDT. Each watched task must 'reset' the TWDT periodically to indicate that they have been allocated CPU time. If a task does not reset within the TWDT timeout period, a warning will be printed with information about which tasks failed to reset the TWDT in time and which tasks are currently running on the ESP32 CPUs and.

The TWDT is built around the Hardware Watchdog Timer in Timer Group 0. The TWDT can be initialized by calling `esp_task_wdt_init()` which will configure the hardware timer. A task can then subscribe to the TWDT using `esp_task_wdt_add()` in order to be watched. Each subscribed task must periodically call `esp_task_wdt_reset()` to reset the TWDT. Failure by any subscribed tasks to periodically call `esp_task_wdt_reset()` indicates that one or more tasks have been starved of CPU time or are stuck in a loop somewhere.

A watched task can be unsubscribed from the TWDT using `esp_task_wdt_delete()`. A task that has been unsubscribed should no longer call `esp_task_wdt_reset()`. Once all tasks have unsubscribed from the TWDT, the TWDT can be deinitialized by calling `esp_task_wdt_deinit()`.

By default `TASK_WDT` in `make menuconfig` will be enabled causing the TWDT to be initialized automatically during startup. Likewise `TASK_WDT_CHECK_IDLE_TASK_CPU0` and `TASK_WDT_CHECK_IDLE_TASK_CPU1` are also enabled by default causing the two Idle Tasks to be subscribed to the TWDT during startup.

## JTAG and watchdogs

While debugging using OpenOCD, the CPUs will be halted every time a breakpoint is reached. However if the watchdog timers continue to run when a breakpoint is encountered, they will eventually trigger a reset making it very difficult to debug code. Therefore OpenOCD will disable the hardware timers of both the interrupt and task watchdogs at every breakpoint. Moreover, OpenOCD will not reenale them upon leaving the breakpoint. This means that interrupt watchdog and task watchdog functionality will essentially be disabled. No warnings or panics from either watchdogs will be generated when the ESP32 is connected to OpenOCD via JTAG.

## Interrupt Watchdog API Reference

### Header File

- `esp32/include/esp_int_wdt.h`

## Functions

void `esp_int_wdt_init()`

Initialize the interrupt watchdog. This is called in the init code if the interrupt watchdog is enabled in menuconfig.

## Task Watchdog API Reference

A full example using the Task Watchdog is available in esp-idf: [system/task\\_watchdog](#)

## Header File

- `esp32/include/esp_task_wdt.h`

## Functions

`esp_err_t esp_task_wdt_init` (uint32\_t *timeout*, bool *panic*)

Initialize the Task Watchdog Timer (TWDT)

This function configures and initializes the TWDT. If the TWDT is already initialized when this function is called, this function will update the TWDT's timeout period and panic configurations instead. After initializing the TWDT, any task can elect to be watched by the TWDT by subscribing to it using `esp_task_wdt_add()`.

### Return

- `ESP_OK`: Initialization was successful
- `ESP_ERR_NO_MEM`: Initialization failed due to lack of memory

**Note** `esp_task_wdt_init()` must only be called after the scheduler started

### Parameters

- `timeout`: Timeout period of TWDT in seconds
- `panic`: Flag that controls whether the panic handler will be executed when the TWDT times out

`esp_err_t esp_task_wdt_deinit` ()

Deinitialize the Task Watchdog Timer (TWDT)

This function will deinitialize the TWDT. Calling this function whilst tasks are still subscribed to the TWDT, or when the TWDT is already deinitialized, will result in an error code being returned.

### Return

- `ESP_OK`: TWDT successfully deinitialized
- `ESP_ERR_INVALID_STATE`: Error, tasks are still subscribed to the TWDT
- `ESP_ERR_NOT_FOUND`: Error, TWDT has already been deinitialized

`esp_err_t esp_task_wdt_add` (*TaskHandle\_t handle*)

Subscribe a task to the Task Watchdog Timer (TWDT)

This function subscribes a task to the TWDT. Each subscribed task must periodically call `esp_task_wdt_reset()` to prevent the TWDT from elapsing its timeout period. Failure to do so will result in a TWDT timeout. If the task being subscribed is one of the Idle Tasks, this function will automatically enable `esp_task_wdt_reset()` to

called from the Idle Hook of the Idle Task. Calling this function whilst the TWDT is uninitialized or attempting to subscribe an already subscribed task will result in an error code being returned.

#### Return

- ESP\_OK: Successfully subscribed the task to the TWDT
- ESP\_ERR\_INVALID\_ARG: Error, the task is already subscribed
- ESP\_ERR\_NO\_MEM: Error, could not subscribe the task due to lack of memory
- ESP\_ERR\_INVALID\_STATE: Error, the TWDT has not been initialized yet

#### Parameters

- `handle`: Handle of the task. Input NULL to subscribe the current running task to the TWDT

#### `esp_err_t esp_task_wdt_reset ()`

Reset the Task Watchdog Timer (TWDT) on behalf of the currently running task.

This function will reset the TWDT on behalf of the currently running task. Each subscribed task must periodically call this function to prevent the TWDT from timing out. If one or more subscribed tasks fail to reset the TWDT on their own behalf, a TWDT timeout will occur. If the IDLE tasks have been subscribed to the TWDT, they will automatically call this function from their idle hooks. Calling this function from a task that has not subscribed to the TWDT, or when the TWDT is uninitialized will result in an error code being returned.

#### Return

- ESP\_OK: Successfully reset the TWDT on behalf of the currently running task
- ESP\_ERR\_NOT\_FOUND: Error, the current running task has not subscribed to the TWDT
- ESP\_ERR\_INVALID\_STATE: Error, the TWDT has not been initialized yet

#### `esp_err_t esp_task_wdt_delete (TaskHandle_t handle)`

Unsubscribes a task from the Task Watchdog Timer (TWDT)

This function will unsubscribe a task from the TWDT. After being unsubscribed, the task should no longer call `esp_task_wdt_reset()`. If the task is an IDLE task, this function will automatically disable the calling of `esp_task_wdt_reset()` from the Idle Hook. Calling this function whilst the TWDT is uninitialized or attempting to unsubscribe an already unsubscribed task from the TWDT will result in an error code being returned.

#### Return

- ESP\_OK: Successfully unsubscribed the task from the TWDT
- ESP\_ERR\_INVALID\_ARG: Error, the task is already unsubscribed
- ESP\_ERR\_INVALID\_STATE: Error, the TWDT has not been initialized yet

#### Parameters

- `handle`: Handle of the task. Input NULL to unsubscribe the current running task.

#### `esp_err_t esp_task_wdt_status (TaskHandle_t handle)`

Query whether a task is subscribed to the Task Watchdog Timer (TWDT)

This function will query whether a task is currently subscribed to the TWDT, or whether the TWDT is initialized.

#### Return :

- ESP\_OK: The task is currently subscribed to the TWDT

- `ESP_ERR_NOT_FOUND`: The task is currently not subscribed to the TWDT
- `ESP_ERR_INVALID_STATE`: The TWDT is not initialized, therefore no tasks can be subscribed

#### Parameters

- `handle`: Handle of the task. Input NULL to query the current running task.

void `esp_task_wdt_feed()`

Reset the TWDT on behalf of the current running task, or subscribe the TWDT to if it has not done so already.

This function is similar to `esp_task_wdt_reset()` and will reset the TWDT on behalf of the current running task. However if this task has not subscribed to the TWDT, this function will automatically subscribe the task. Therefore, an unsubscribed task will subscribe to the TWDT on its first call to this function, then proceed to reset the TWDT on subsequent calls of this function.

**Warning** This function is deprecated, use `esp_task_wdt_add()` and `esp_task_wdt_reset()` instead

## 2.7.7 Inter-Processor Call

### Overview

Due to the dual core nature of the ESP32, there are instances where a certain function must be run in the context of a particular core (e.g. allocating ISR to an interrupt source of a particular core). The IPC (Inter-Processor Call) feature allows for the execution of functions on a particular CPU.

A given function can be executed on a particular core by calling `esp_ipc_call()` or `esp_ipc_call_blocking()`. IPC is implemented via two high priority FreeRTOS tasks pinned to each CPU known as the IPC Tasks. The two IPC Tasks remain inactive (blocked) until `esp_ipc_call()` or `esp_ipc_call_blocking()` is called. When an IPC Task of a particular core is unblocked, it will preempt the current running task on that core and execute a given function.

### Usage

`esp_ipc_call()` unblocks the IPC task on a particular core to execute a given function. The task that calls `esp_ipc_call()` will be blocked until the IPC Task begins execution of the given function. `esp_ipc_call_blocking()` is similar but will block the calling task until the IPC Task has completed execution of the given function.

Functions executed by IPCs must be functions of type `void func(void *arg)`. To run more complex functions which require a larger stack, the IPC tasks' stack size can be configured by modifying `IPC_TASK_STACK_SIZE` in `menuconfig`. The IPC API is protected by a mutex hence simultaneous IPC calls are not possible.

Care should be taken to avoid deadlock when writing functions to be executed by IPC, especially when attempting to take a mutex within the function.

### API Reference

#### Header File

- `esp32/include/esp_ipc.h`

## Functions

`esp_err_t esp_ipc_call` (`uint32_t cpu_id`, `esp_ipc_func_t func`, `void *arg`)

Execute a function on the given CPU.

Run a given function on a particular CPU. The given function must accept a `void*` argument and return `void`. The given function is run in the context of the IPC task of the CPU specified by the `cpu_id` parameter. The calling task will be blocked until the IPC task begins executing the given function. If another IPC call is ongoing, the calling task will block until the other IPC call completes. The stack size allocated for the IPC task can be configured in the “Inter-Processor Call (IPC) task stack size” setting in `menuconfig`. Increase this setting if the given function requires more stack than default.

**Note** In single-core mode, returns `ESP_ERR_INVALID_ARG` for `cpu_id` 1.

### Return

- `ESP_ERR_INVALID_ARG` if `cpu_id` is invalid
- `ESP_ERR_INVALID_STATE` if the FreeRTOS scheduler is not running
- `ESP_OK` otherwise

### Parameters

- `cpu_id`: CPU where the given function should be executed (0 or 1)
- `func`: Pointer to a function of type `void func(void* arg)` to be executed
- `arg`: Arbitrary argument of type `void*` to be passed into the function

`esp_err_t esp_ipc_call_blocking` (`uint32_t cpu_id`, `esp_ipc_func_t func`, `void *arg`)

Execute a function on the given CPU and blocks until it completes.

Run a given function on a particular CPU. The given function must accept a `void*` argument and return `void`. The given function is run in the context of the IPC task of the CPU specified by the `cpu_id` parameter. The calling task will be blocked until the IPC task completes execution of the given function. If another IPC call is ongoing, the calling task will block until the other IPC call completes. The stack size allocated for the IPC task can be configured in the “Inter-Processor Call (IPC) task stack size” setting in `menuconfig`. Increase this setting if the given function requires more stack than default.

**Note** In single-core mode, returns `ESP_ERR_INVALID_ARG` for `cpu_id` 1.

### Return

- `ESP_ERR_INVALID_ARG` if `cpu_id` is invalid
- `ESP_ERR_INVALID_STATE` if the FreeRTOS scheduler is not running
- `ESP_OK` otherwise

### Parameters

- `cpu_id`: CPU where the given function should be executed (0 or 1)
- `func`: Pointer to a function of type `void func(void* arg)` to be executed
- `arg`: Arbitrary argument of type `void*` to be passed into the function

## 2.7.8 High Resolution Timer

### Overview

Although FreeRTOS provides software timers, these timers have a few limitations:

- Maximum resolution is equal to RTOS tick period
- Timer callbacks are dispatched from a low-priority task

Hardware timers are free from both of the limitations, but often they are less convenient to use. For example, application components may need timer events to fire at certain times in the future, but the hardware timer only contains one “compare” value used for interrupt generation. This means that some facility needs to be built on top of the hardware timer to manage the list of pending events can dispatch the callbacks for these events as corresponding hardware interrupts happen.

`esp_timer` set of APIs provide such facility. Internally, `esp_timer` uses a 32-bit hardware timer (FRC1, “legacy” timer). `esp_timer` provides one-shot and periodic timers, microsecond time resolution, and 64-bit range.

Timer callbacks are dispatched from a high-priority `esp_timer` task. Because all the callbacks are dispatched from the same task, it is recommended to only do the minimal possible amount of work from the callback itself, posting an event to a lower priority task using a queue instead.

### Using `esp_timer` APIs

Single timer is represented by `esp_timer_handle_t` type. Timer has a callback function associated with it. This callback function is called from the `esp_timer` task each time the timer elapses.

- To create a timer, call `esp_timer_create()`.
- To delete the timer when it is no longer needed, call `esp_timer_delete()`.

The timer can be started in one-shot mode or in periodic mode.

- To start the timer in one-shot mode, call `esp_timer_start_once()`, passing the time interval after which the callback should be called. When the callback gets called, the timer is considered to be stopped.
- To start the timer in periodic mode, call `esp_timer_start_periodic()`, passing the period with which the callback should be called. The timer keeps running until `esp_timer_stop()` is called.

Note that the timer must not be running when `esp_timer_start_once()` or `esp_timer_start_periodic()` is called. To restart a running timer, call `esp_timer_stop()` first, then call one of the start functions.

### Obtaining Current Time

`esp_timer` also provides a convenience function to obtain the time passed since start-up, with microsecond precision: `esp_timer_get_time()`. This function returns the number of microseconds since `esp_timer` was initialized, which usually happens shortly before `app_main` function is called.

Unlike `gettimeofday` function, values returned by `esp_timer_get_time()`:

- Start from zero after the chip wakes up from deep sleep
- Do not have timezone or DST adjustments applied

## API Reference

### Header File

- `esp32/include/esp_timer.h`

### Functions

`esp_err_t esp_timer_init ()`  
Initialize `esp_timer` library.

**Note** This function is called from startup code. Applications do not need to call this function before using other `esp_timer` APIs.

#### Return

- `ESP_OK` on success
- `ESP_ERR_NO_MEM` if allocation has failed
- `ESP_ERR_INVALID_STATE` if already initialized
- other errors from interrupt allocator

`esp_err_t esp_timer_deinit ()`  
De-initialize `esp_timer` library.

**Note** Normally this function should not be called from applications

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if not yet initialized

`esp_err_t esp_timer_create (const esp_timer_create_args_t *create_args, esp_timer_handle_t *out_handle)`  
Create an `esp_timer` instance.

**Note** When done using the timer, delete it with `esp_timer_delete` function.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if some of the `create_args` are not valid
- `ESP_ERR_INVALID_STATE` if `esp_timer` library is not initialized yet
- `ESP_ERR_NO_MEM` if memory allocation fails

#### Parameters

- `create_args`: Pointer to a structure with timer creation arguments. Not saved by the library, can be allocated on the stack.
- `out_handle`: Output, pointer to `esp_timer_handle_t` variable which will hold the created timer handle.

`esp_err_t esp_timer_start_once` (*esp\_timer\_handle\_t* timer, `uint64_t` timeout\_us)  
Start one-shot timer.

Timer should not be running when this function is called.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if the handle is invalid
- ESP\_ERR\_INVALID\_STATE if the timer is already running

**Parameters**

- timer: timer handle created using `esp_timer_create`
- timeout\_us: timer timeout, in microseconds relative to the current moment

`esp_err_t esp_timer_start_periodic` (*esp\_timer\_handle\_t* timer, `uint64_t` period)  
Start a periodic timer.

Timer should not be running when this function is called. This function will start the timer which will trigger every 'period' microseconds.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if the handle is invalid
- ESP\_ERR\_INVALID\_STATE if the timer is already running

**Parameters**

- timer: timer handle created using `esp_timer_create`
- period: timer period, in microseconds

`esp_err_t esp_timer_stop` (*esp\_timer\_handle\_t* timer)  
Stop the timer.

This function stops the timer previously started using `esp_timer_start_once` or `esp_timer_start_periodic`.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if the timer is not running

**Parameters**

- timer: timer handle created using `esp_timer_create`

`esp_err_t esp_timer_delete` (*esp\_timer\_handle\_t* timer)  
Delete an `esp_timer` instance.

The timer must be stopped before deleting. A one-shot timer which has expired does not need to be stopped.

**Return**

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if the timer is not running

### Parameters

- `timer`: timer handle allocated using `esp_timer_create`

`int64_t esp_timer_get_time ()`

Get time in microseconds since boot.

**Return** number of microseconds since `esp_timer_init` was called (this normally happens early during application startup).

`esp_err_t esp_timer_dump (FILE *stream)`

Dump the list of timers to a stream.

If `CONFIG_ESP_TIMER_PROFILING` option is enabled, this prints the list of all the existing timers. Otherwise, only the list active timers is printed.

The format is:

name period alarm times\_armed times\_triggered total\_callback\_run\_time

where:

name — timer name (if `CONFIG_ESP_TIMER_PROFILING` is defined), or timer pointer period — period of timer, in microseconds, or 0 for one-shot timer alarm - time of the next alarm, in microseconds since boot, or 0 if the timer is not started

The following fields are printed if `CONFIG_ESP_TIMER_PROFILING` is defined:

times\_armed — number of times the timer was armed via `esp_timer_start_X` times\_triggered - number of times the callback was called total\_callback\_run\_time - total time taken by callback to execute, across all calls

### Return

- `ESP_OK` on success
- `ESP_ERR_NO_MEM` if can not allocate temporary buffer for the output

### Parameters

- `stream`: stream (such as `stdout`) to dump the information to

## Structures

`struct esp_timer_create_args_t`

Timer configuration passed to `esp_timer_create`.

### Public Members

`esp_timer_cb_t callback`

Function to call when timer expires.

`void *arg`

Argument to pass to the callback.

`esp_timer_dispatch_t dispatch_method`

Call the callback from task or from ISR.

`const char *name`

Timer name, used in `esp_timer_dump` function.

## Type Definitions

**typedef struct esp\_timer \*esp\_timer\_handle\_t**

Opaque type representing a single esp\_timer.

**typedef void (\*esp\_timer\_cb\_t)(void \*arg)**

Timer callback function type.

### Parameters

- `arg`: pointer to opaque user-specific data

## Enumerations

**enum esp\_timer\_dispatch\_t**

Method for dispatching timer callback.

*Values:*

**ESP\_TIMER\_TASK**

Callback is called from timer task.

## 2.7.9 Logging library

### Overview

Log library has two ways of managing log verbosity: compile time, set via menuconfig; and runtime, using `esp_log_level_set` function.

At compile time, filtering is done using `CONFIG_LOG_DEFAULT_LEVEL` macro, set via menuconfig. All logging statements for levels higher than `CONFIG_LOG_DEFAULT_LEVEL` will be removed by the preprocessor.

At run time, all logs below `CONFIG_LOG_DEFAULT_LEVEL` are enabled by default. `esp_log_level_set` function may be used to set logging level per module. Modules are identified by their tags, which are human-readable ASCII zero-terminated strings.

### How to use this library

In each C file which uses logging functionality, define TAG variable like this:

```
static const char* TAG = "MyModule";
```

then use one of logging macros to produce output, e.g:

```
ESP_LOGW(TAG, "Baud rate error %.1f%%. Requested: %d baud, actual: %d baud", error * 100, baud_req, baud_real);
```

Several macros are available for different verbosity levels:

- `ESP_LOGE` - error
- `ESP_LOGW` - warning
- `ESP_LOGI` - info
- `ESP_LOGD` - debug

- ESP\_LOGV - verbose

Additionally there is an `_EARLY_` variant for each of these macros (e.g. `ESP_EARLY_LOGE`). These variants can run in startup code, before heap allocator and syscalls have been initialized. When compiling bootloader, normal `ESP_LOGx` macros fall back to the same implementation as `ESP_EARLY_LOGx` macros. So the only place where `ESP_EARLY_LOGx` have to be used explicitly is the early startup code, such as heap allocator initialization code.

(Note that such distinction would not have been necessary if we would have an `ets_vprintf` function in the ROM. Then it would be possible to switch implementation from `_EARLY_` version to normal version on the fly. Unfortunately, `ets_vprintf` in ROM has been inlined by the compiler into `ets_printf`, so it is not accessible outside.)

To override default verbosity level at file or component scope, define `LOG_LOCAL_LEVEL` macro. At file scope, define it before including `esp_log.h`, e.g.:

```
#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include "esp_log.h"
```

At component scope, define it in component makefile:

```
CFLAGS += -D LOG_LOCAL_LEVEL=ESP_LOG_DEBUG
```

To configure logging output per module at runtime, add calls to `esp_log_level_set` function:

```
esp_log_level_set("*", ESP_LOG_ERROR);           // set all components to ERROR level
esp_log_level_set("wifi", ESP_LOG_WARN);        // enable WARN logs from WiFi stack
esp_log_level_set("dhcpc", ESP_LOG_INFO);       // enable INFO logs from DHCP client
```

## Logging to Host via JTAG

By default logging library uses `vprintf`-like function to write formatted output to dedicated UART. With calling a simple API, all log output may be routed to JTAG instead, and make the logging several times faster. For details please refer to section *Logging to Host*.

## Application Example

Log library is commonly used by most of `esp-idf` components and examples. For demonstration of log functionality check `examples` folder of `espressif/esp-idf` repository, that among others, contains the following examples:

- `system/ota`
- `storage/sd_card`
- `protocols/https_request`

## API Reference

### Header File

- `log/include/esp_log.h`

## Functions

void **esp\_log\_level\_set** (**const** char \*tag, *esp\_log\_level\_t* level)

Set log level for given tag.

If logging for given component has already been enabled, changes previous setting.

### Parameters

- **tag**: Tag of the log entries to enable. Must be a non-NULL zero terminated string. Value "\*" resets log level for all tags to the given value.
- **level**: Selects log level to enable. Only logs at this and lower levels will be shown.

*vprintf\_like\_t* **esp\_log\_set\_vprintf** (*vprintf\_like\_t* func)

Set function used to output log entries.

By default, log output goes to UART0. This function can be used to redirect log output to some other destination, such as file or network. Returns the original log handler, which may be necessary to return output to the previous destination.

**Return** func old Function used for output.

### Parameters

- **func**: new Function used for output. Must have same signature as vprintf.

uint32\_t **esp\_log\_timestamp** (void)

Function which returns timestamp to be used in log output.

This function is used in expansion of ESP\_LOGx macros. In the 2nd stage bootloader, and at early application startup stage this function uses CPU cycle counter as time source. Later when FreeRTOS scheduler start running, it switches to FreeRTOS tick count.

For now, we ignore millisecond counter overflow.

**Return** timestamp, in milliseconds

uint32\_t **esp\_log\_early\_timestamp** (void)

Function which returns timestamp to be used in log output.

This function uses HW cycle counter and does not depend on OS, so it can be safely used after application crash.

**Return** timestamp, in milliseconds

void **esp\_log\_write** (*esp\_log\_level\_t* level, **const** char \*tag, **const** char \*format, ...)

Write message into the log.

This function is not intended to be used directly. Instead, use one of ESP\_LOGE, ESP\_LOGW, ESP\_LOGI, ESP\_LOGD, ESP\_LOGV macros.

This function or these macros should not be used from an interrupt.

## Macros

**ESP\_LOG\_BUFFER\_HEX\_LEVEL** (tag, buffer, buff\_len, level)

Log a buffer of hex bytes at specified level, separated into 16 bytes each line.

**Parameters**

- tag: description tag
- buffer: Pointer to the buffer array
- buff\_len: length of buffer in bytes
- level: level of the log

**ESP\_LOG\_BUFFER\_CHAR\_LEVEL** (tag, buffer, buff\_len, level)

Log a buffer of characters at specified level, separated into 16 bytes each line. Buffer should contain only printable characters.

**Parameters**

- tag: description tag
- buffer: Pointer to the buffer array
- buff\_len: length of buffer in bytes
- level: level of the log

**ESP\_LOG\_BUFFER\_HEXDUMP** (tag, buffer, buff\_len, level)

Dump a buffer to the log at specified level.

The dump log shows just like the one below:

```

W (195) log_example: 0x3ffb4280  45 53 50 33 32 20 69 73  20 67 72 65 61 74 2c
↪20 |ESP32 is great, |
W (195) log_example: 0x3ffb4290  77 6f 72 6b 69 6e 67 20  61 6c 6f 6e 67 20 77
↪69 |working along wi|
W (205) log_example: 0x3ffb42a0  74 68 20 74 68 65 20 49  44 46 2e 00
↪ |th the IDF..|

```

It is highly recommend to use terminals with over 102 text width.

**Parameters**

- tag: description tag
- buffer: Pointer to the buffer array
- buff\_len: length of buffer in bytes
- level: level of the log

**ESP\_LOG\_BUFFER\_HEX** (tag, buffer, buff\_len)

Log a buffer of hex bytes at Info level.

See `esp_log_buffer_hex_level`

**Parameters**

- tag: description tag
- buffer: Pointer to the buffer array
- buff\_len: length of buffer in bytes

**ESP\_LOG\_BUFFER\_CHAR** (tag, buffer, buff\_len)

Log a buffer of characters at Info level. Buffer should contain only printable characters.

See `esp_log_buffer_char_level`

**Parameters**

- `tag`: description tag
- `buffer`: Pointer to the buffer array
- `buff_len`: length of buffer in bytes

**`esp_log_buffer_hex`**

**`esp_log_buffer_char`**

**`LOG_COLOR_E`**

**`LOG_COLOR_W`**

**`LOG_COLOR_I`**

**`LOG_COLOR_D`**

**`LOG_COLOR_V`**

**`LOG_RESET_COLOR`**

**`LOG_FORMAT`** (letter, format)

**`LOG_LOCAL_LEVEL`**

**`ESP_EARLY_LOGE`** (tag, format, ...)

macro to output logs in startup code, before heap allocator and syscalls have been initialized. log at `ESP_LOG_ERROR` level.

See `printf,ESP_LOGE`

**`ESP_EARLY_LOGW`** (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_WARN` level.

See `ESP_EARLY_LOGE,ESP_LOGE,printf`

**`ESP_EARLY_LOGI`** (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_INFO` level.

See `ESP_EARLY_LOGE,ESP_LOGE,printf`

**`ESP_EARLY_LOGD`** (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_DEBUG` level.

See `ESP_EARLY_LOGE,ESP_LOGE,printf`

**`ESP_EARLY_LOGV`** (tag, format, ...)

macro to output logs in startup code at `ESP_LOG_VERBOSE` level.

See `ESP_EARLY_LOGE,ESP_LOGE,printf`

**`ESP_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 speicified level.

See `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 speicified level. Also check the level with `LOG_LOCAL_LEVEL`.

See `printf`, `ESP_LOG_LEVEL`

## Type Definitions

```
typedef int (*vprintf_like_t)(const char *, va_list)
```

## Enumerations

```
enum esp_log_level_t
```

Log level.

*Values:*

**ESP\_LOG\_NONE**

No log output

**ESP\_LOG\_ERROR**

Critical errors, software module can not recover on its own

**ESP\_LOG\_WARN**

Error conditions from which recovery measures have been taken

**ESP\_LOG\_INFO**

Information messages which describe normal flow of events

**ESP\_LOG\_DEBUG**

Extra information which is not necessary for normal use (values, pointers, sizes, etc).

**ESP\_LOG\_VERBOSE**

Bigger chunks of debugging information, or frequent messages which can potentially flood the output.

## 2.7.10 Application Level Tracing

## Overview

IDF provides useful feature for program behaviour analysis: application level tracing. It is implemented in the corresponding library and can be enabled via menuconfig. This feature allows to transfer arbitrary data between host and ESP32 via JTAG interface with small overhead on program execution. Developers can use this library to send application specific state of execution to the host and receive commands or other type of info in the opposite direction at runtime. The main use cases of this library are:

1. Collecting application specific data, see *Application Specific Tracing*
2. Lightweight logging to the host, see *Logging to Host*
3. System behaviour analysis, see *System Behaviour Analysis with SEGGER SystemView*

## API Reference

### Header File

- `app_trace/include/esp_app_trace.h`

### Functions

`esp_err_t esp_appttrace_init ()`  
Initializes application tracing module.

**Note** Should be called before any `esp_appttrace_xxx` call.

**Return** `ESP_OK` on success, otherwise see `esp_err_t`

`void esp_appttrace_down_buffer_config (uint8_t *buf, uint32_t size)`  
Configures down buffer.

**Note** Needs to be called before initiating any data transfer using `esp_appttrace_buffer_get` and `esp_appttrace_write`. This function does not protect internal data by lock.

#### Parameters

- `buf`: Address of buffer to use for down channel (host to target) data.
- `size`: Size of the buffer.

`uint8_t *esp_appttrace_buffer_get (esp_appttrace_dest_t dest, uint32_t size, uint32_t tmo)`  
Allocates buffer for trace data. After data in buffer are ready to be sent off `esp_appttrace_buffer_put` must be called to indicate it.

**Return** non-NULL on success, otherwise NULL.

#### Parameters

- `dest`: Indicates HW interface to send data.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_apprtrace_buffer_put (esp_apprtrace_dest_t dest, uint8_t *ptr, uint32_t tmo)`

Indicates that the data in buffer are ready to be sent off. This function is a counterpart of and must be preceded by `esp_apprtrace_buffer_get`.

**Return** ESP\_OK on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to send data. Should be identical to the same parameter in call to `esp_apprtrace_buffer_get`.
- `ptr`: Address of trace buffer to release. Should be the value returned by call to `esp_apprtrace_buffer_get`.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_apprtrace_write (esp_apprtrace_dest_t dest, const void *data, uint32_t size, uint32_t tmo)`

Writes data to trace buffer.

**Return** ESP\_OK on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to send data.
- `data`: Address of data to write to trace buffer.
- `size`: Size of data to write to trace buffer.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`int esp_apprtrace_vprintf_to (esp_apprtrace_dest_t dest, uint32_t tmo, const char *fmt, va_list ap)`  
vprintf-like function to sent log messages to host via specified HW interface.

**Return** Number of bytes written.

#### Parameters

- `dest`: Indicates HW interface to send data.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.
- `fmt`: Address of format string.
- `ap`: List of arguments.

`int esp_apprtrace_vprintf (const char *fmt, va_list ap)`  
vprintf-like function to sent log messages to host.

**Return** Number of bytes written.

#### Parameters

- `fmt`: Address of format string.
- `ap`: List of arguments.

`esp_err_t esp_apprtrace_flush (esp_apprtrace_dest_t dest, uint32_t tmo)`

Flushes remaining data in trace buffer to host.

**Return** ESP\_OK on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to flush data on.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_appttrace_flush_nolock (esp_appttrace_dest_t dest, uint32_t min_sz, uint32_t tmo)`

Flushes remaining data in trace buffer to host without locking internal data. This is special version of `esp_appttrace_flush` which should be called from panic handler.

**Return** `ESP_OK` on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to flush data on.
- `min_sz`: Threshold for flushing data. If current filling level is above this value, data will be flushed. TRAX destinations only.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_appttrace_read (esp_appttrace_dest_t dest, void *data, uint32_t *size, uint32_t tmo)`

Reads host data from trace buffer.

**Return** `ESP_OK` on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to read the data on.
- `data`: Address of buffer to put data from trace buffer.
- `size`: Pointer to store size of read data. Before call to this function pointed memory must hold requested size of data
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`uint8_t *esp_appttrace_down_buffer_get (esp_appttrace_dest_t dest, uint32_t *size, uint32_t tmo)`

Retrieves incoming data buffer if any. After data in buffer are processed `esp_appttrace_down_buffer_put` must be called to indicate it.

**Return** non-NULL on success, otherwise NULL.

#### Parameters

- `dest`: Indicates HW interface to receive data.
- `size`: Address to store size of available data in down buffer. Must be initialized with requested value.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

`esp_err_t esp_appttrace_down_buffer_put (esp_appttrace_dest_t dest, uint8_t *ptr, uint32_t tmo)`

Indicates that the data in down buffer are processed. This function is a counterpart of and must be preceded by `esp_appttrace_down_buffer_get`.

**Return** `ESP_OK` on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to receive data. Should be identical to the same parameter in call to `esp_appttrace_down_buffer_get`.

- `ptr`: Address of trace buffer to release. Should be the value returned by call to `esp_appttrace_down_buffer_get`.
- `tmo`: Timeout for operation (in us). Use `ESP_APPTRACE_TMO_INFINITE` to wait indefinitely.

bool **esp\_appttrace\_host\_is\_connected** (*esp\_appttrace\_dest\_t dest*)

Checks whether host is connected.

**Return** true if host is connected, otherwise false

**Parameters**

- `dest`: Indicates HW interface to use.

void \***esp\_appttrace\_fopen** (*esp\_appttrace\_dest\_t dest*, const char \**path*, const char \**mode*)

Opens file on host. This function has the same semantic as 'fopen' except for the first argument.

**Return** non zero file handle on success, otherwise 0

**Parameters**

- `dest`: Indicates HW interface to use.
- `path`: Path to file.
- `mode`: Mode string. See fopen for details.

int **esp\_appttrace\_fclose** (*esp\_appttrace\_dest\_t dest*, void \**stream*)

Closes file on host. This function has the same semantic as 'fclose' except for the first argument.

**Return** Zero on success, otherwise non-zero. See fclose for details.

**Parameters**

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by `esp_appttrace_fopen`.

size\_t **esp\_appttrace\_fwrite** (*esp\_appttrace\_dest\_t dest*, const void \**ptr*, size\_t *size*, size\_t *nmemb*, void \**stream*)

Writes to file on host. This function has the same semantic as 'fwrite' except for the first argument.

**Return** Number of written items. See fwrite for details.

**Parameters**

- `dest`: Indicates HW interface to use.
- `ptr`: Address of data to write.
- `size`: Size of an item.
- `nmemb`: Number of items to write.
- `stream`: File handle returned by `esp_appttrace_fopen`.

size\_t **esp\_appttrace\_fread** (*esp\_appttrace\_dest\_t dest*, void \**ptr*, size\_t *size*, size\_t *nmemb*, void \**stream*)

Read file on host. This function has the same semantic as 'fread' except for the first argument.

**Return** Number of read items. See fread for details.

**Parameters**

- `dest`: Indicates HW interface to use.
- `ptr`: Address to store read data.
- `size`: Size of an item.
- `nmemb`: Number of items to read.
- `stream`: File handle returned by `esp_appttrace_fopen`.

int **esp\_appttrace\_fseek** (*esp\_appttrace\_dest\_t* *dest*, void \**stream*, long *offset*, int *whence*)

Set position indicator in file on host. This function has the same semantic as 'fseek' except for the first argument.

**Return** Zero on success, otherwise non-zero. See `fseek` for details.

#### Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by `esp_appttrace_fopen`.
- `offset`: Offset. See `fseek` for details.
- `whence`: Position in file. See `fseek` for details.

int **esp\_appttrace\_ftell** (*esp\_appttrace\_dest\_t* *dest*, void \**stream*)

Get current position indicator for file on host. This function has the same semantic as 'ftell' except for the first argument.

**Return** Current position in file. See `ftell` for details.

#### Parameters

- `dest`: Indicates HW interface to use.
- `stream`: File handle returned by `esp_appttrace_fopen`.

int **esp\_appttrace\_fstop** (*esp\_appttrace\_dest\_t* *dest*)

Indicates to the host that all file operations are completed. This function should be called after all file operations are finished and indicate to the host that it can perform cleanup operations (close open files etc.).

**Return** `ESP_OK` on success, otherwise see `esp_err_t`

#### Parameters

- `dest`: Indicates HW interface to use.

void **esp\_gcov\_dump** (void)

Triggers gcov info dump. This function waits for the host to connect to target before dumping data.

## Enumerations

enum **esp\_appttrace\_dest\_t**

Application trace data destinations bits.

*Values:*

**ESP\_APPTRACE\_DEST\_TRAX** = 0x1  
JTAG destination.

`ESP_APPTRACE_DEST_UART0 = 0x2`  
UART destination.

## 2.7.11 Power Management

### Overview

Power management algorithm included in ESP-IDF can adjust APB frequency, CPU frequency, and put the chip into light sleep mode to run the 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 instance, a driver for a peripheral clocked from APB can request the APB frequency to be set to 80 MHz, for the duration while the peripheral is used. Another example is that the RTOS will request the CPU to run at the highest configured frequency while there are tasks ready to run. Yet another example is a peripheral driver which needs interrupts to be enabled. Such driver can request light sleep to be disabled.

Naturally, requesting higher APB or CPU frequency or disabling light sleep causes higher current consumption. Components should try to limit usage of power management locks to the shortest amount of time possible.

### Configuration

Power management can be enabled at compile time, using `PM_ENABLE` option.

Enabling power management features comes at the cost of increased interrupt latency. Extra latency depends on a number of factors, among which are CPU frequency, single/dual core mode, whether frequency switch needs to be performed or not. Minimal extra latency is 0.2us (when CPU frequency is 240MHz, and frequency scaling is not enabled), maximum extra latency is 40us (when frequency scaling is enabled, and a switch from 40MHz to 80MHz is performed on interrupt entry).

Dynamic frequency scaling (DFS) can be enabled in the application by calling `esp_pm_configure()` function. Its argument is a structure defining frequency scaling settings (for ESP32, minimum and maximum CPU frequencies). Alternatively, `PM_DFS_INIT_AUTO` option can be enabled in menuconfig. If enabled, maximal CPU frequency is determined by `ESP32_DEFAULT_CPU_FREQ_MHZ` setting, and minimal CPU frequency is set to the XTAL frequency.

---

**Note:** `esp_pm_configure()` function also has provisions for enabling automatic light sleep mode. However this feature is not fully supported yet, so `esp_pm_configure` will return an `ESP_ERR_NOT_SUPPORTED` if automatic light sleep is requested.

---

### Power Management Locks

As mentioned in the overview, applications can acquire/release locks to control the power management algorithm. When application takes a lock, power management algorithm operation is restricted in a way described below, for each lock. When the lock is released, such restriction is removed.

Different parts of the application can take the same lock. In this case, lock must be released the same number of times as it was acquired, in order for power management algorithm to resume.

In ESP32, three types of locks are supported:

**ESP\_PM\_CPU\_FREQ\_MAX** Requests CPU frequency to be at the maximal value set via `esp_pm_configure()`. For ESP32, this value can be set to 80, 160, or 240MHz.

**ESP\_PM\_APB\_FREQ\_MAX** Requests APB frequency to be at the maximal supported value. For ESP32, this is 80 MHz.

**ESP\_PM\_NO\_LIGHT\_SLEEP** Prevents automatic light sleep from being used. Note: currently taking this lock has no effect, as automatic light sleep is never used.

## Power Management Algorithm for the ESP32

When dynamic frequency scaling is enabled, CPU frequency will be switched as follows:

- If maximal CPU frequency (set using `esp_pm_configure()` or `ESP32_DEFAULT_CPU_FREQ_MHZ`) is 240 MHz:
  1. When `ESP_PM_CPU_FREQ_MAX` or `ESP_PM_APB_FREQ_MAX` locks are acquired, CPU frequency will be 240 MHz, and APB frequency will be 80 MHz.
  2. Otherwise, frequency will be switched to the minimal value set using `esp_pm_configure()` (usually, XTAL).
- If maximal CPU frequency is 160 MHz:
  1. When `ESP_PM_CPU_FREQ_MAX` is acquired, CPU frequency is set to 160 MHz, and APB frequency to 80 MHz.
  2. When `ESP_PM_CPU_FREQ_MAX` is not acquired, but `ESP_PM_APB_FREQ_MAX` is, CPU and APB frequencies are set to 80 MHz.
  3. Otherwise, frequency will be switched to the minimal value set using `esp_pm_configure()` (usually, XTAL).
- If maximal CPU frequency is 80 MHz:
  1. When `ESP_PM_CPU_FREQ_MAX` or `ESP_PM_APB_FREQ_MAX` locks are acquired, CPU and APB frequencies will be 80 MHz.
  2. Otherwise, frequency will be switched to the minimal value set using `esp_pm_configure()` (usually, XTAL).

## Dynamic Frequency Scaling and Peripheral Drivers

When DFS is enabled, APB frequency can be changed several times within a single RTOS tick. Some peripherals can work normally even when APB frequency changes; some can not.

The following peripherals can work even when APB frequency is changing:

- UART: if `REF_TICK` is used as clock source (see `use_ref_tick` member of `uart_config_t`).
- LEDC: if `REF_TICK` is used as clock source (see `ledc_timer_config()` function).
- RMT: if `REF_TICK` is used as clock source. Currently the driver does not support `REF_TICK`, but it can be enabled by clearing `RMT_REF_ALWAYS_ON_CHx` bit for the respective channel.

Currently, the following peripheral drivers are aware of DFS and will use `ESP_PM_APB_FREQ_MAX` lock for the duration of the transaction:

- SPI master
- SDMMC

The following drivers will hold `ESP_PM_APB_FREQ_MAX` lock while the driver is enabled:

- SPI slave — between calls to `spi_slave_initialize()` and `cpp:func:spi_slave_free`.

- Ethernet — between calls to `esp_eth_enable()` and `esp_eth_disable()`.
- WiFi — between calls to `esp_wifi_start()` and `esp_wifi_stop()`. If modem sleep is enabled, lock will be released for the periods of time when radio is disabled.
- Bluetooth — between calls to `esp_bt_controller_enable()` and `esp_bt_controller_disable()`.

The following peripheral drivers are not aware of DFS yet. Applications need to acquire/release locks when necessary:

- I2C
- I2S
- MCPWM
- PCNT
- Sigma-delta
- Timer group

## API Reference

### Header File

- `esp32/include/esp_pm.h`

### Functions

`esp_err_t esp_pm_configure(const void *config)`  
Set implementation-specific power management configuration.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if the configuration values are not correct
- `ESP_ERR_NOT_SUPPORTED` if certain combination of values is not supported, or if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

#### Parameters

- `config`: pointer to implementation-specific configuration structure (e.g. `esp_pm_config_esp32`)

`esp_err_t esp_pm_lock_create(esp_pm_lock_type_t lock_type, int arg, const char *name, esp_pm_lock_handle_t *out_handle)`

Initialize a lock handle for certain power management parameter.

When lock is created, initially it is not taken. Call `esp_pm_lock_acquire` to take the lock.

This function must not be called from an ISR.

#### Return

- `ESP_OK` on success
- `ESP_ERR_NO_MEM` if the lock structure can not be allocated
- `ESP_ERR_INVALID_ARG` if `out_handle` is `NULL` or type argument is not valid

- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

#### Parameters

- `lock_type`: Power management constraint which the lock should control
- `arg`: argument, value depends on `lock_type`, see `esp_pm_lock_type_t`
- `name`: arbitrary string identifying the lock (e.g. “wifi” or “spi”). Used by the `esp_pm_dump_locks` function to list existing locks. May be set to `NULL`. If not set to `NULL`, must point to a string which is valid for the lifetime of the lock.
- `out_handle`: handle returned from this function. Use this handle when calling `esp_pm_lock_delete`, `esp_pm_lock_acquire`, `esp_pm_lock_release`. Must not be `NULL`.

`esp_err_t` **`esp_pm_lock_acquire`** (*`esp_pm_lock_handle_t` handle*)

Take a power management lock.

Once the lock is taken, power management algorithm will not switch to the mode specified in a call to `esp_pm_lock_create`, or any of the lower power modes (higher numeric values of ‘mode’).

The lock is recursive, in the sense that if `esp_pm_lock_acquire` is called a number of times, `esp_pm_lock_release` has to be called the same number of times in order to release the lock.

This function may be called from an ISR.

This function is not thread-safe w.r.t. calls to other `esp_pm_lock_*` functions for the same handle.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if the handle is invalid
- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

#### Parameters

- `handle`: handle obtained from `esp_pm_lock_create` function

`esp_err_t` **`esp_pm_lock_release`** (*`esp_pm_lock_handle_t` handle*)

Release the lock taken using `esp_pm_lock_acquire`.

Call to this functions removes power management restrictions placed when taking the lock.

Locks are recursive, so if `esp_pm_lock_acquire` is called a number of times, `esp_pm_lock_release` has to be called the same number of times in order to actually release the lock.

This function may be called from an ISR.

This function is not thread-safe w.r.t. calls to other `esp_pm_lock_*` functions for the same handle.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if the handle is invalid
- `ESP_ERR_INVALID_STATE` if lock is not acquired
- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

#### Parameters

- `handle`: handle obtained from `esp_pm_lock_create` function

`esp_err_t esp_pm_lock_delete(esp_pm_lock_handle_t handle)`

Delete a lock created using `esp_pm_lock`.

The lock must be released before calling this function.

This function must not be called from an ISR.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if the handle argument is `NULL`
- `ESP_ERR_INVALID_STATE` if the lock is still acquired
- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

#### Parameters

- `handle`: handle obtained from `esp_pm_lock_create` function

`esp_err_t esp_pm_dump_locks(FILE *stream)`

Dump the list of all locks to `stderr`

This function dumps debugging information about locks created using `esp_pm_lock_create` to an output stream.

This function must not be called from an ISR. If `esp_pm_lock_acquire/release` are called while this function is running, inconsistent results may be reported.

#### Return

- `ESP_OK` on success
- `ESP_ERR_NOT_SUPPORTED` if `CONFIG_PM_ENABLE` is not enabled in `sdkconfig`

#### Parameters

- `stream`: stream to print information to; use `stdout` or `stderr` to print to the console; use `fmemopen/open_memstream` to print to a string buffer.

## Type Definitions

```
typedef struct esp_pm_lock *esp_pm_lock_handle_t
```

Opaque handle to the power management lock.

## Enumerations

```
enum esp_pm_lock_type_t
```

Power management constraints.

*Values:*

```
ESP_PM_CPU_FREQ_MAX
```

Require CPU frequency to be at the maximum value set via `esp_pm_configure`. Argument is unused and should be set to 0.

```
ESP_PM_APB_FREQ_MAX
```

Require APB frequency to be at the maximum value supported by the chip. Argument is unused and should be set to 0.

### **ESP\_PM\_NO\_LIGHT\_SLEEP**

Prevent the system from going into light sleep. Argument is unused and should be set to 0.

## Header File

- `esp32/include/esp32/pm.h`

## Structures

### **struct esp\_pm\_config\_esp32\_t**

Power management config for ESP32.

Pass a pointer to this structure as an argument to `esp_pm_configure` function.

### Public Members

`rtc_cpu_freq_t max_cpu_freq`

Maximum CPU frequency to use

`rtc_cpu_freq_t min_cpu_freq`

Minimum CPU frequency to use when no frequency locks are taken

`bool light_sleep_enable`

Enter light sleep when no locks are taken

## 2.7.12 Sleep Modes

### Overview

ESP32 is capable of light sleep and deep sleep power saving modes.

In light sleep mode, digital peripherals, most of the RAM, and CPUs are clock-gated, and supply voltage is reduced. Upon exit from light sleep, peripherals and CPUs resume operation, their internal state is preserved.

In deep sleep mode, CPUs, most of the RAM, and all the digital peripherals which are clocked from `APB_CLK` are powered off. The only parts of the chip which can still be powered on are: RTC controller, RTC peripherals (including ULP coprocessor), and RTC memories (slow and fast).

Wakeup from deep and light sleep modes can be done using several sources. These sources can be combined, in this case the chip will wake up when any one of the sources is triggered. Wakeup sources can be enabled using `esp_sleep_enable_X_wakeup` APIs. Next section describes these APIs in detail. Wakeup sources can be configured at any moment before entering light or deep sleep mode.

Additionally, the application can force specific powerdown modes for the RTC peripherals and RTC memories using `esp_sleep_pd_config` API.

Once wakeup sources are configured, application can enter sleep mode using `esp_light_sleep_start` or `esp_deep_sleep_start` APIs. At this point the hardware will be configured according to the requested wakeup sources, and RTC controller will either power down or power off the CPUs and digital peripherals.

## WiFi/BT and sleep modes

In deep sleep mode, wireless peripherals are powered down. Before entering sleep mode, applications must disable WiFi and BT using appropriate calls (`esp_bluedroid_disable`, `esp_bt_controller_disable`, `esp_wifi_stop`).

WiFi can coexist with light sleep mode, allowing the chip to go into light sleep mode when there is no network activity, and waking up the chip from light sleep mode when required. However **APIs described in this section can not be used for that purpose**. `esp_light_sleep_start` forces the chip to enter light sleep mode, regardless of whether WiFi is active or not. Automatic entry into light sleep mode, coordinated with WiFi driver, will be supported using a separate set of APIs.

## Wakeup sources

### Timer

RTC controller has a built in timer which can be used to wake up the chip after a predefined amount of time. Time is specified at microsecond precision, but the actual resolution depends on the clock source selected for RTC SLOW\_CLK. See chapter “Reset and Clock” of the ESP32 Technical Reference Manual for details about RTC clock options.

This wakeup mode doesn’t require RTC peripherals or RTC memories to be powered on during sleep.

The following function can be used to enable deep sleep wakeup using a timer.

```
esp_err_t esp_sleep_enable_timer_wakeup (uint64_t time_in_us)
    Enable wakeup by timer.
```

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if value is out of range (TBD)

#### Parameters

- `time_in_us`: time before wakeup, in microseconds

## Touch pad

RTC IO module contains logic to trigger wakeup when a touch sensor interrupt occurs. You need to configure the touch pad interrupt before the chip starts deep sleep.

Revisions 0 and 1 of the ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e. `ESP_PD_DOMAIN_RTC_PERIPH` should be set to `ESP_PD_OPTION_AUTO`).

```
esp_err_t esp_sleep_enable_touchpad_wakeup ()
    Enable wakeup by touch sensor.
```

**Note** In revisions 0 and 1 of the 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.

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_STATE if wakeup triggers conflict

## External wakeup (ext0)

RTC IO module contains logic to trigger wakeup when one of RTC GPIOs is set to a predefined logic level. RTC IO is part of RTC peripherals power domain, so RTC peripherals will be kept powered on during deep sleep if this wakeup source is requested.

Because RTC IO module is enabled in this mode, internal pullup or pulldown resistors can also be used. They need to be configured by the application using `rtc_gpio_pullup_en` and `rtc_gpio_pulldown_en` functions, before calling `esp_sleep_start`.

In revisions 0 and 1 of the ESP32, this wakeup source is incompatible with ULP and touch wakeup sources.

**Warning:** After wake up from sleep, IO pad used for wakeup will be configured as RTC IO. Before using this pad as digital GPIO, reconfigure it using `rtc_gpio_deinit (gpio_num)` function.

`esp_err_t esp_sleep_enable_ext0_wakeup (gpio_num_t gpio_num, int level)`

Enable wakeup using a pin.

This function uses external wakeup feature of RTC\_IO peripheral. It will work only if RTC peripherals are kept on during sleep.

This feature can monitor any pin which is an RTC IO. Once the pin transitions into the state given by level argument, the chip will be woken up.

**Note** This function does not modify pin configuration. The pin is configured in `esp_sleep_start`, immediately before entering sleep mode.

**Note** In revisions 0 and 1 of the ESP32, ext0 wakeup source can not be used together with touch or ULP wakeup sources.

### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if the selected GPIO is not an RTC GPIO, or the mode is invalid
- ESP\_ERR\_INVALID\_STATE if wakeup triggers conflict

### Parameters

- `gpio_num`: GPIO number used as wakeup source. Only GPIOs which are have RTC functionality can be used: 0,2,4,12-15,25-27,32-39.
- `level`: input level which will trigger wakeup (0=low, 1=high)

## External wakeup (ext1)

RTC controller contains logic to trigger wakeup using multiple RTC GPIOs. One of the two logic functions can be used to trigger wakeup:

- wake up if any of the selected pins is high (ESP\_EXT1\_WAKEUP\_ANY\_HIGH)
- wake up if all the selected pins are low (ESP\_EXT1\_WAKEUP\_ALL\_LOW)

This wakeup source is implemented by the RTC controller. As such, RTC peripherals and RTC memories can be powered down in this mode. However, if RTC peripherals are powered down, internal pullup and pulldown resistors will be disabled. To use internal pullup or pulldown resistors, request RTC peripherals power domain to be kept on during sleep, and configure pullup/pulldown resistors using `rtc_gpio_` functions, before entering sleep:

```
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_ON);
gpio_pullup_dis(gpio_num);
gpiopulldown_en(gpio_num);
```

**Warning:** After wake up from sleep, IO pad(s) used for wakeup will be configured as RTC IO. Before using these pads as digital GPIOs, reconfigure them using `rtc_gpio_deinit(gpio_num)` function.

The following function can be used to enable this wakeup mode:

```
esp_err_t esp_sleep_enable_ext1_wakeup(uint64_t mask, esp_sleep_ext1_wakeup_mode_t mode)
```

Enable wakeup using multiple pins.

This function uses external wakeup feature of RTC controller. It will work even if RTC peripherals are shut down during sleep.

This feature can monitor any number of pins which are in RTC IOs. Once any of the selected pins goes into the state given by mode argument, the chip will be woken up.

**Note** This function does not modify pin configuration. The pins are configured in `esp_sleep_start`, immediately before entering sleep mode.

**Note** internal pullups and pulldowns don't work when RTC peripherals are shut down. In this case, external resistors need to be added. Alternatively, RTC peripherals (and pullups/pulldowns) may be kept enabled using `esp_sleep_pd_config` function.

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if any of the selected GPIOs is not an RTC GPIO, or mode is invalid

#### Parameters

- `mask`: bit mask of GPIO numbers which will cause wakeup. Only GPIOs which have RTC functionality can be used in this bit map: 0,2,4,12-15,25-27,32-39.
- `mode`: select logic function used to determine wakeup condition:
  - ESP\_EXT1\_WAKEUP\_ALL\_LOW: wake up when all selected GPIOs are low
  - ESP\_EXT1\_WAKEUP\_ANY\_HIGH: wake up when any of the selected GPIOs is high

```
enum esp_sleep_ext1_wakeup_mode_t
```

Logic function used for EXT1 wakeup mode.

*Values:*

```
ESP_EXT1_WAKEUP_ALL_LOW = 0
```

Wake the chip when all selected GPIOs go low.

```
ESP_EXT1_WAKEUP_ANY_HIGH = 1
```

Wake the chip when any of the selected GPIOs go high.

## ULP coprocessor wakeup

ULP coprocessor can run while the chip is in sleep mode, and may be used to poll sensors, monitor ADC or touch sensor values, and wake up the chip when a specific event is detected. ULP coprocessor is part of RTC peripherals power domain, and it runs the program stored in RTC slow memory. RTC slow memory will be powered on during

sleep if this wakeup mode is requested. RTC peripherals will be automatically powered on before ULP coprocessor starts running the program; once the program stops running, RTC peripherals are automatically powered down again.

Revisions 0 and 1 of the ESP32 only support this wakeup mode when RTC peripherals are not forced to be powered on (i.e. `ESP_PD_DOMAIN_RTC_PERIPH` should be set to `ESP_PD_OPTION_AUTO`).

The following function can be used to enable this wakeup mode:

```
esp_err_t esp_sleep_enable_ulp_wakeup ()  
    Enable wakeup by ULP coprocessor.
```

**Note** In revisions 0 and 1 of the ESP32, ULP 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.

**Return**

- `ESP_OK` on success
- `ESP_ERR_INVALID_STATE` if ULP co-processor is not enabled or if wakeup triggers conflict

### Power-down of RTC peripherals and memories

By default, `esp_deep_sleep_start` and `esp_light_sleep_start` functions will power down all RTC power domains which are not needed by the enabled wakeup sources. To override this behaviour, `esp_sleep_pd_config` function is provided.

Note: in revision 0 of the ESP32, RTC fast memory will always be kept enabled in deep sleep, so that the deep sleep stub can run after reset. This can be overridden, if the application doesn't need clean reset behaviour after deep sleep.

If some variables in the program are placed into RTC slow memory (for example, using `RTC_DATA_ATTR` attribute), RTC slow memory will be kept powered on by default. This can be overridden using `esp_sleep_pd_config` function, if desired.

```
esp_err_t esp_sleep_pd_config (esp_sleep_pd_domain_t domain, esp_sleep_pd_option_t option)  
    Set power down mode for an RTC power domain in sleep mode.
```

If not set using this API, all power domains default to `ESP_PD_OPTION_AUTO`.

**Return**

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if either of the arguments is out of range

**Parameters**

- `domain`: power domain to configure
- `option`: power down option (`ESP_PD_OPTION_OFF`, `ESP_PD_OPTION_ON`, or `ESP_PD_OPTION_AUTO`)

```
enum esp_sleep_pd_domain_t
```

Power domains which can be powered down in sleep mode.

*Values:*

```
ESP_PD_DOMAIN_RTC_PERIPH  
    RTC IO, sensors and ULP co-processor.
```

```
ESP_PD_DOMAIN_RTC_SLOW_MEM  
    RTC slow memory.
```

**ESP\_PD\_DOMAIN\_RTC\_FAST\_MEM**

RTC fast memory.

**ESP\_PD\_DOMAIN\_MAX**

Number of domains.

**enum esp\_sleep\_pd\_option\_t**

Power down options.

*Values:*

**ESP\_PD\_OPTION\_OFF**

Power down the power domain in sleep mode.

**ESP\_PD\_OPTION\_ON**

Keep power domain enabled during sleep mode.

**ESP\_PD\_OPTION\_AUTO**

Keep power domain enabled in sleep mode, if it is needed by one of the wakeup options. Otherwise power it down.

## Entering light sleep

The following function can be used to enter light sleep once wakeup sources are configured. It is also possible to go into light sleep with no wakeup sources configured, in this case the chip will be in light sleep mode indefinitely, until external reset is applied.

`esp_err_t esp_light_sleep_start ()`

Enter light sleep with the configured wakeup options.

**Return**

- ESP\_OK on success (returned after wakeup)
- ESP\_ERR\_INVALID\_STATE if WiFi or BT is not stopped

## Entering deep sleep

The following function can be used to enter deep sleep once wakeup sources are configured. It is also possible to go into deep sleep with no wakeup sources configured, in this case the chip will be in deep sleep mode indefinitely, until external reset is applied.

`void esp_deep_sleep_start ()`

Enter deep sleep with the configured wakeup options.

This function does not return.

## Checking sleep wakeup cause

The following function can be used to check which wakeup source has triggered wakeup from sleep mode. For touch pad and ext1 wakeup sources, it is possible to identify pin or touch pad which has caused wakeup.

`esp_sleep_wakeup_cause_t esp_sleep_get_wakeup_cause ()`

Get the source which caused wakeup from sleep.

**Return** wakeup cause, or ESP\_DEEP\_SLEEP\_WAKEUP\_UNDEFINED if reset happened for reason other than deep sleep wakeup

**enum esp\_sleep\_wakeup\_cause\_t**

Sleep wakeup cause.

*Values:*

**ESP\_SLEEP\_WAKEUP\_UNDEFINED**

**ESP\_SLEEP\_WAKEUP\_EXT0**

In case of deep sleep, reset was not caused by exit from deep sleep.

**ESP\_SLEEP\_WAKEUP\_EXT1**

Wakeup caused by external signal using RTC\_IO.

**ESP\_SLEEP\_WAKEUP\_TIMER**

Wakeup caused by external signal using RTC\_CNTL.

**ESP\_SLEEP\_WAKEUP\_TOUCHPAD**

Wakeup caused by timer.

**ESP\_SLEEP\_WAKEUP\_ULP**

Wakeup caused by touchpad.

*touch\_pad\_t* **esp\_sleep\_get\_touchpad\_wakeup\_status()**

Get the touch pad which caused wakeup.

If wakeup was caused by another source, this function will return TOUCH\_PAD\_MAX;

**Return** touch pad which caused wakeup

*uint64\_t* **esp\_sleep\_get\_ext1\_wakeup\_status()**

Get the bit mask of GPIOs which caused wakeup (ext1)

If wakeup was caused by another source, this function will return 0.

**Return** bit mask, if GPIO<sub>n</sub> caused wakeup, BIT(n) will be set

## Application Example

Implementation of basic functionality of deep sleep is shown in [protocols/sntp](#) example, where ESP module is periodically waken up to retrieve time from NTP server.

More extensive example in [system/deep\\_sleep](#) illustrates usage of various deep sleep wakeup triggers and ULP coprocessor programming.

## 2.7.13 Base MAC address

### Overview

Several MAC addresses (universally administered by IEEE) are uniquely assigned to the networking interfaces (WiFi/BT/Ethernet). The final octet of each universally administered MAC address increases by one. Only the first one which is called base MAC address of them is stored in EFUSE or external storage, the others are generated from it. Here, 'generate' means adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address.

If the universally administered MAC addresses are not enough for all of the networking interfaces. Local administered MAC addresses which are derived from universally administered MAC addresses are assigned to the reset of networking interfaces.

A definition of local vs universal MAC address can be found on [Wikipedia](#).

The number of universally administered MAC address can be configured using `make menuconfig`.

## Base MAC address

If using the default base MAC address factory programmed by Espressif in BLK0 of EFUSE, nothing needs to be done.

If using a custom base MAC address stored in BLK3 of EFUSE, call API `esp_efuse_mac_get_custom()` to get the base MAC address which is stored in BLK3 of EFUSE. If correct MAC address is returned, then call `esp_base_mac_addr_set()` to set the base MAC address for system to generate the MAC addresses used by the networking interfaces(WiFi/BT/Ethernet). There are 192 bits storage spaces for custom to store base MAC address in BLK3 of EFUSE. They are `EFUSE_BLK3_RDATA0`, `EFUSE_BLK3_RDATA1`, `EFUSE_BLK3_RDATA2`, `EFUSE_BLK3_RDATA3`, `EFUSE_BLK3_RDATA4` and `EFUSE_BLK3_RDATA5`, each of them is 32 bits register. The format of the 192 bits storage spaces is:

Field	Bits	Range	Description
version	8	[[191:184]	1: useful. 0: useless
reserve	112	[[183:72]	reserved
mac address	64	[[71:8]	base MAC address
mac crc	8	[[7:0]	crc of base MAC address

If using base MAC address stored in external storage, firstly get the base MAC address stored in external storage, then call API `esp_base_mac_addr_set()` to set the base MAC address for system to generate the MAC addresses used by the networking interfaces(WiFi/BT/Ethernet).

All of the steps must be done before initializing the networking interfaces(WiFi/BT/Ethernet). It is recommended to do it in `app_main()` which can be referenced in [system/base\\_mac\\_address](#).

## Number of universally administered MAC address

If the number of universal MAC addresses is two, only two interfaces (WiFi station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (WiFi softap and Ethernet) receive local MAC addresses. These are derived from the universal WiFi station and Bluetooth MAC addresses, respectively.

If the number of universal MAC addresses is four, all four interfaces (WiFi station, WiFi softap, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address.

When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

## API Reference

### Header Files

- `esp32/include/esp_system.h`

## Functions

`esp_err_t esp_base_mac_addr_set (uint8_t *mac)`

Set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage e.g. flash and EEPROM.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. If using base MAC address stored in BLK3 of EFUSE or external storage, call this API to set base MAC address with the MAC address which is stored in BLK3 of EFUSE or external storage before initializing WiFi/BT/Ethernet.

**Return** ESP\_OK on success

### Parameters

- `mac`: base MAC address, length: 6 bytes.

`esp_err_t esp_efuse_mac_get_custom (uint8_t *mac)`

Return base MAC address which was previously written to BLK3 of EFUSE.

Base MAC address is used to generate the MAC addresses used by the networking interfaces. This API returns the custom base MAC address which was previously written to BLK3 of EFUSE. Writing this EFUSE allows setting of a different (non-Espressif) base MAC address. It is also possible to store a custom base MAC address elsewhere, see `esp_base_mac_addr_set()` for details.

**Return** ESP\_OK on success ESP\_ERR\_INVALID\_VERSION An invalid MAC version field was read from BLK3 of EFUSE ESP\_ERR\_INVALID\_CRC An invalid MAC CRC was read from BLK3 of EFUSE

### Parameters

- `mac`: base MAC address, length: 6 bytes.

## 2.7.14 Over The Air Updates (OTA)

### OTA Process Overview

The OTA update mechanism allows a device to update itself based on data received while the normal firmware is running (for example, over WiFi or Bluetooth.)

OTA requires configuring the *Partition Table* of the device with at least two “OTA app slot” partitions (ie `ota_0` and `ota_1`) and an “OTA Data Partition”.

The OTA operation functions write a new app firmware image to whichever OTA app slot is not currently being used for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

### OTA Data Partition

An OTA data partition (type `data`, subtype `ota`) must be included in the *Partition Table* of any project which uses the OTA functions.

For factory boot settings, the OTA data partition should contain no data (all bytes erased to 0xFF). In this case the esp-idf software bootloader will boot the factory app if it is present in the the partition table. If no factory app is included in the partition table, the first available OTA slot (usually `ota_0`) is booted.

After the first OTA update, the OTA data partition is updated to specify which OTA app slot partition should be booted next.

The OTA data partition is two flash sectors (0x2000 bytes) in size, to prevent problems if there is a power failure while it is being written. Sectors are independently erased and written with matching data, and if they disagree a counter field is used to determine which sector was written more recently.

### See also

- *Partition Table documentation*
- *Lower-Level SPI Flash/Partition API*

### Application Example

End-to-end example of OTA firmware update workflow: [system/ota](#).

### API Reference

#### Header File

- `app_update/include/esp_ota_ops.h`

#### Functions

`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.

#### Return

- `ESP_OK`: OTA operation commenced successfully.
- `ESP_ERR_INVALID_ARG`: partition or out\_handle arguments were NULL, or partition doesn't point to an OTA app partition.
- `ESP_ERR_NO_MEM`: Cannot allocate memory for OTA operation.
- `ESP_ERR_OTA_PARTITION_CONFLICT`: Partition holds the currently running firmware, cannot update in place.
- `ESP_ERR_NOT_FOUND`: Partition argument not found in partition table.
- `ESP_ERR_OTA_SELECT_INFO_INVALID`: The OTA data partition contains invalid data.
- `ESP_ERR_INVALID_SIZE`: Partition doesn't fit in configured flash size.
- `ESP_ERR_FLASH_OP_TIMEOUT` or `ESP_ERR_FLASH_OP_FAIL`: Flash write failed.

#### Parameters

- `partition`: Pointer to info for partition which will receive the OTA update. Required.

- `image_size`: Size of new OTA app image. Partition will be erased in order to receive this size of image. If 0 or `OTA_SIZE_UNKNOWN`, the entire partition is erased.
- `out_handle`: On success, returns a handle which should be used for subsequent `esp_ota_write()` and `esp_ota_end()` calls.

`esp_err_t esp_ota_write(esp_ota_handle_t handle, const void *data, size_t size)`

Write OTA update data to partition.

This function can be called multiple times as data is received during the OTA operation. Data is written sequentially to the partition.

#### Return

- `ESP_OK`: Data was written to flash successfully.
- `ESP_ERR_INVALID_ARG`: handle is invalid.
- `ESP_ERR_OTA_VALIDATE_FAILED`: First byte of image contains invalid app image magic byte.
- `ESP_ERR_FLASH_OP_TIMEOUT` or `ESP_ERR_FLASH_OP_FAIL`: Flash write failed.
- `ESP_ERR_OTA_SELECT_INFO_INVALID`: OTA data partition has invalid contents

#### Parameters

- `handle`: Handle obtained from `esp_ota_begin`
- `data`: Data buffer to write
- `size`: Size of data buffer in bytes.

`esp_err_t esp_ota_end(esp_ota_handle_t handle)`

Finish OTA update and validate newly written app image.

**Note** After calling `esp_ota_end()`, the handle is no longer valid and any memory associated with it is freed (regardless of result).

#### Return

- `ESP_OK`: Newly written OTA app image is valid.
- `ESP_ERR_NOT_FOUND`: OTA handle was not found.
- `ESP_ERR_INVALID_ARG`: Handle was never written to.
- `ESP_ERR_OTA_VALIDATE_FAILED`: OTA image is invalid (either not a valid app image, or - if secure boot is enabled - signature failed to verify.)
- `ESP_ERR_INVALID_STATE`: If flash encryption is enabled, this result indicates an internal error writing the final encrypted bytes to flash.

#### Parameters

- `handle`: Handle obtained from `esp_ota_begin()`.

`esp_err_t esp_ota_set_boot_partition(const esp_partition_t *partition)`

Configure OTA data for a new boot partition.

**Note** If this function returns `ESP_OK`, calling `esp_restart()` will boot the newly configured app partition.

#### Return

- `ESP_OK`: OTA data updated, next reboot will use specified partition.

- `ESP_ERR_INVALID_ARG`: partition argument was NULL or didn't point to a valid OTA partition of type "app".
- `ESP_ERR_OTA_VALIDATE_FAILED`: Partition contained invalid app image. Also returned if secure boot is enabled and signature validation failed.
- `ESP_ERR_NOT_FOUND`: OTA data partition not found.
- `ESP_ERR_FLASH_OP_TIMEOUT` or `ESP_ERR_FLASH_OP_FAIL`: Flash erase or write failed.

#### Parameters

- `partition`: Pointer to info for partition containing app image to boot.

**const** *esp\_partition\_t* \***esp\_ota\_get\_boot\_partition**(void)

Get partition info of currently configured boot app.

If `esp_ota_set_boot_partition()` has been called, the partition which was set by that function will be returned.

If `esp_ota_set_boot_partition()` has not been called, the result is usually the same as `esp_ota_get_running_partition()`. The two results are not equal if the configured boot partition does not contain a valid app (meaning that the running partition will be an app that the bootloader chose via fallback).

If the OTA data partition is not present or not valid then the result is the first app partition found in the partition table. In priority order, this means: the factory app, the first OTA app slot, or the test app partition.

Note that there is no guarantee the returned partition is a valid app. Use `esp_image_load(ESP_IMAGE_VERIFY, ...)` to verify if the returned partition contains a bootable image.

**Return** Pointer to info for partition structure, or NULL if partition table is invalid or a flash read operation failed. Any returned pointer is valid for the lifetime of the application.

**const** *esp\_partition\_t* \***esp\_ota\_get\_running\_partition**(void)

Get partition info of currently running app.

This function is different to `esp_ota_get_boot_partition()` in that it ignores any change of selected boot partition caused by `esp_ota_set_boot_partition()`. Only the app whose code is currently running will have its partition information returned.

The partition returned by this function may also differ from `esp_ota_get_boot_partition()` if the configured boot partition is somehow invalid, and the bootloader fell back to a different app partition at boot.

**Return** Pointer to info for partition structure, or NULL if no partition is found or flash read operation failed. Returned pointer is valid for the lifetime of the application.

**const** *esp\_partition\_t* \***esp\_ota\_get\_next\_update\_partition**(**const** *esp\_partition\_t* \**start\_from*)

Return the next OTA app partition which should be written with a new firmware.

Call this function to find an OTA app partition which can be passed to `esp_ota_begin()`.

Finds next partition round-robin, starting from the current running partition.

**Return** Pointer to info for partition which should be updated next. NULL result indicates invalid OTA data partition, or that no eligible OTA app slot partition was found.

#### Parameters

- `start_from`: If set, treat this partition info as describing the current running partition. Can be NULL, in which case `esp_ota_get_running_partition()` is used to find the currently running partition. The result of this function is never the same as this argument.

## Macros

### **OTA\_SIZE\_UNKNOWN**

Used for `esp_ota_begin()` if new image size is unknown

### **ESP\_ERR\_OTA\_BASE**

Base error code for `ota_ops` api

### **ESP\_ERR\_OTA\_PARTITION\_CONFLICT**

Error if request was to write or erase the current running partition

### **ESP\_ERR\_OTA\_SELECT\_INFO\_INVALID**

Error if OTA data partition contains invalid content

### **ESP\_ERR\_OTA\_VALIDATE\_FAILED**

Error if OTA app image is invalid

## 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()`.

Example code for this API section is provided in `system` directory of ESP-IDF examples.

## 2.8 Configuration Options

### 2.8.1 Introduction

ESP-IDF uses `Kconfig` system to provide a compile-time configuration mechanism. `Kconfig` is based around options of several types: integer, string, boolean. `Kconfig` files specify dependencies between options, default values of the options, the way the options are grouped together, etc.

Applications developers can use `make menuconfig` build target to edit components' configuration. This configuration is saved inside `sdkconfig` file in the project root directory. Based on `sdkconfig`, application build targets will generate `sdkconfig.h` file in the build directory, and will make `sdkconfig` options available to component makefiles.

### 2.8.2 Using `sdkconfig.defaults`

When updating ESP-IDF version, it is not uncommon to find that new `Kconfig` options are introduced. When this happens, application build targets will offer an interactive prompt to select values for the new options. New values are then written into `sdkconfig` file. To suppress interactive prompts, applications can either define `BATCH_BUILD` environment variable, which will cause all prompts to be suppressed. This is the same effect as that of `V` or `VERBOSE` variables. Alternatively, `defconfig` build target can be used to update configuration for all new variables to the default values.

In some cases, such as when `sdkconfig` file is under revision control, the fact that `sdkconfig` file gets changed by the build system may be inconvenient. The build system offers a way to avoid this, in the form of `sdkconfig.defaults` file. This file is never touched by the build system, and must be created manually. It can contain all the options which matter for the given application. The format is the same as that of the `sdkconfig` file. Once `sdkconfig.defaults` is created, `sdkconfig` can be deleted and added to the ignore list of the revision control system (e.g. `.gitignore` file for `git`). Project build targets will automatically create `sdkconfig` file, populated

with the settings from `sdkconfig.defaults` file, and the rest of the settings will be set to their default values. Note that when `make defconfig` is used, settings in `sdkconfig` will be overridden by the ones in `sdkconfig.defaults`. For more information, see *Custom sdkconfig defaults*.

### 2.8.3 Configuration Options Reference

Subsequent sections contain the list of available ESP-IDF options, automatically generated from Kconfig files. Note that depending on the options selected, some options listed here may not be visible by default in the interface of `menuconfig`.

By convention, all option names are upper case with underscores. When Kconfig generates `sdkconfig` and `sdkconfig.h` files, option names are prefixed with `CONFIG_`. So if an option `ENABLE_FOO` is defined in a Kconfig file and selected in `menuconfig`, then `sdkconfig` and `sdkconfig.h` files will have `CONFIG_ENABLE_FOO` defined. In this reference, option names are also prefixed with `CONFIG_`, same as in the source code.

#### LWIP

##### L2\_TO\_L3\_COPY

Enable copy between Layer2 and Layer3 packets

*Found in: Component config > LWIP*

If this feature is enabled, all traffic from layer2(WIFI Driver) will be copied to a new buffer before sending it to layer3(LWIP stack), freeing the layer2 buffer. Please be notified that the total layer2 receiving buffer is fixed and ESP32 currently supports 25 layer2 receiving buffer, when layer2 buffer runs out of memory, then the incoming packets will be dropped in hardware. The layer3 buffer is allocated from the heap, so the total layer3 receiving buffer depends on the available heap size, when heap runs out of memory, no copy will be sent to layer3 and packet will be dropped in layer2. Please make sure you fully understand the impact of this feature before enabling it.

##### 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.

##### 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`.

##### LWIP\_SO\_REUSE\_RXTOALL

`SO_REUSEADDR` copies broadcast/multicast to all matches

*Found in: Component config > LWIP*

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.

## LWIP\_SO\_RCVBUF

Enable `SO_RCVBUF` option

*Found in: Component config > LWIP*

Enabling this option allows checking for available data on a netconn.

## LWIP\_DHCP\_MAX\_NTP\_SERVERS

Maximum number of NTP servers

*Found in: Component config > LWIP*

Set maximum number of NTP servers used by LwIP SNTP module. First argument of `sntp_setserver/sntp_setservername` functions is limited to this value.

## LWIP\_IP\_FRAG

Enable fragment outgoing IP packets

*Found in: Component config > LWIP*

Enabling this option allows fragmenting outgoing IP packets if their size exceeds MTU.

## LWIP\_IP\_REASSEMBLY

Enable reassembly incoming fragmented IP packets

*Found in: Component config > LWIP*

Enabling this option allows reassembling incoming fragmented IP packets.

## LWIP\_STATS

Enable LWIP statistics

*Found in: Component config > LWIP*

Enabling this option allows LWIP statistics

## LWIP\_ETHARP\_TRUST\_IP\_MAC

Enable LWIP ARP trust

*Found in: Component config > LWIP*

Enabling this option allows ARP table to be updated.

If this option is enabled, the incoming IP packets cause the ARP table to be updated with the source MAC and IP addresses supplied in the packet. You may want to disable this if you do not trust LAN peers to have the correct addresses, or as a limited approach to attempt to handle spoofing. If disabled, lwIP will need to make a new ARP request if the peer is not already in the ARP table, adding a little latency. The peer *is* in the ARP table if it requested our address before. Also notice that this slows down input processing of every IP packet!

**There are two known issues in real application if this feature is enabled:**

- The LAN peer may have bug to update the ARP table after the ARP entry is aged out. If the ARP entry on the LAN peer is aged out but failed to be updated, all IP packets sent from LWIP to the LAN peer will be dropped by LAN peer.
- The LAN peer may not be trustful, the LAN peer may send IP packets to LWIP with two different MACs, but the same IP address. If this happens, the LWIP has problem to receive IP packets from LAN peer.

So the recommendation is to disable this option. Here the LAN peer means the other side to which the ESP station or soft-AP is connected.

## ESP\_GRATUITOUS\_ARP

Send gratuitous ARP periodically

*Found in: Component config > LWIP*

Enable this option allows to send gratuitous ARP periodically.

This option solve the compatibility issues.If the ARP table of the AP is old, and the AP doesn't send ARP request to update it's ARP table, this will lead to the STA sending IP packet fail. Thus we send gratuitous ARP periodically to let AP update it's ARP table.

## GARP\_TMR\_INTERVAL

GARP timer interval(seconds)

*Found in: Component config > LWIP*

Set the timer interval for gratuitous ARP. The default value is 60s

## TCPIP\_RECVMBOX\_SIZE

TCPIP task receive mail box size

*Found in: Component config > LWIP*

Set TCPIP task receive mail box size. Generally bigger value means higher throughput but more memory. The value should be bigger than UDP/TCP mail box size.

## 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.

## DHCP server

### LWIP\_DHCPS\_LEASE\_UNIT

Multiplier for lease time, in seconds

*Found in: Component config > LWIP > DHCP server*

The DHCP server is calculating lease time multiplying the sent and received times by this number of seconds per unit. The default is 60, that equals one minute.

### LWIP\_DHCPS\_MAX\_STATION\_NUM

Maximum number of stations

*Found in: Component config > LWIP > DHCP server*

The maximum number of DHCP clients that are connected to the server. After this number is exceeded, DHCP server removes of the oldest device from it's address pool, without notification.

### 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.

### LWIP\_AUTOIP\_TRIES

DHCP Probes before self-assigning IPv4 LL address

*Found in: Component config > LWIP*

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.)

### LWIP\_AUTOIP\_MAX\_CONFLICTS

Max IP conflicts before rate limiting

*Found in: Component config > LWIP*

If the AUTOIP functionality detects this many IP conflicts while self-assigning an address, it will go into a rate limited mode.

## LWIP\_AUTOIP\_RATE\_LIMIT\_INTERVAL

Rate limited interval (seconds)

*Found in: Component config > LWIP*

If rate limiting self-assignment requests, wait this long between each request.

## 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.

## LWIP\_LOOPBACK\_MAX\_PBUFS

Max queued loopback packets per interface

*Found in: Component config > LWIP*

Configure the maximum number of packets which can be queued for loopback on a given interface. Reducing this number may cause packets to be dropped, but will avoid filling memory with queued packet data.

## TCP

### 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.

### 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.

## TCP\_MAXRTX

Maximum number of retransmissions of data segments

*Found in: Component config > LWIP > TCP*

Set maximum number of retransmissions of data segments.

## TCP\_SYNMAXRTX

Maximum number of retransmissions of SYN segments

*Found in: Component config > LWIP > TCP*

Set maximum number of retransmissions of SYN segments.

## 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 1436 will give best throughput.

## TCP\_MSL

Maximum segment lifetime (MSL)

*Found in: Component config > LWIP > TCP*

Set maximum segment lifetime in in milliseconds.

## 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.

## 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.

## TCP\_RECVMBOX\_SIZE

Default TCP receive mail box size

*Found in: Component config > LWIP > TCP*

Set TCP receive mail box size. Generally bigger value means higher throughput but more memory. The recommended value is:  $TCP\_WND\_DEFAULT/TCP\_MSS + 2$ , e.g. if  $TCP\_WND\_DEFAULT=14360$ ,  $TCP\_MSS=1436$ , then the recommended receive mail box size is  $(14360/1436 + 2) = 12$ .

TCP receive mail box is a per socket mail box, when the application receives packets from TCP socket, LWIP core firstly posts the packets to TCP receive mail box and the application then fetches the packets from mail box. It means LWIP can cache maximum `TCP_RECVMBOX_SIZE` packets for each TCP socket, so the maximum possible cached TCP packets for all TCP sockets is `TCP_RECVMBOX_SIZE` multiplies the maximum TCP socket number. In other words, the bigger `TCP_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 TCP receive mail box is big enough to avoid packet drop between LWIP core and application.

## 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.

## ESP\_TCP\_KEEP\_CONNECTION\_WHEN\_IP\_CHANGES

Keep TCP connections when IP changed

*Found in: Component config > LWIP > TCP*

This option is enabled when the following scenario happen: network dropped and reconnected, IP changes is like:  $192.168.0.2 \rightarrow 0.0.0.0 \rightarrow 192.168.0.2$

Disable this option to keep consistent with the original LWIP code behavior.

## 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:**

- TCP\_OVERSIZE\_MSS
- TCP\_OVERSIZE\_QUARTER\_MSS
- TCP\_OVERSIZE\_DISABLE

## UDP

### 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.

### 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 cache maximum UDP\_RECVMBOX\_SIZE packets for each UDP socket, so the maximum possible cached UDP packets for all UDP sockets is UDP\_RECVMBOX\_SIZE multiplies the maximum UDP socket number. In other words, the bigger UDP\_RECVMBOX\_SIZE means more memory. On the other hand, if the receive mail box is too small, the mail box may be full. If the mail box is full, the LWIP drops the packets. So generally we need to make sure the UDP receive mail box is big enough to avoid packet drop between LWIP core and application.

### 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.

### PPP\_SUPPORT

Enable PPP support (new/experimental)

*Found in: Component config > LWIP*

Enable PPP stack. Now only PPP over serial is possible.

PPP over serial support is experimental and unsupported.

### **PPP\_PAP\_SUPPORT**

Enable PAP support

*Found in: Component config > LWIP*

Enable Password Authentication Protocol (PAP) support

### **PPP\_CHAP\_SUPPORT**

Enable CHAP support

*Found in: Component config > LWIP*

Enable Challenge Handshake Authentication Protocol (CHAP) support

### **PPP\_MSCHAP\_SUPPORT**

Enable MSCHAP support

*Found in: Component config > LWIP*

Enable Microsoft version of the Challenge-Handshake Authentication Protocol (MSCHAP) support

### **PPP\_MPPE\_SUPPORT**

Enable MPPE support

*Found in: Component config > LWIP*

Enable Microsoft Point-to-Point Encryption (MPPE) support

### **PPP\_DEBUG\_ON**

Enable PPP debug log output

*Found in: Component config > LWIP*

Enable PPP debug log output

### **ICMP**

#### **LWIP\_MULTICAST\_PING**

Respond to multicast pings

*Found in: Component config > LWIP > ICMP*

## LWIP\_BROADCAST\_PING

Respond to broadcast pings

*Found in: Component config > LWIP > ICMP*

## LWIP\_RAW\_API

### 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.

## libsodium

### LIBSODIUM\_USE\_MBEDTLS\_SHA

Use mbedTLS SHA256 & SHA512 implementations

*Found in: Component config > libsodium*

If this option is enabled, libsodium will use thin wrappers around mbedTLS for SHA256 & SHA512 operations.

This saves some code size if mbedTLS is also used. However it is incompatible with hardware SHA acceleration (due to the way libsodium's API manages SHA state).

## FreeRTOS

### FREERTOS\_UNICORE

Run FreeRTOS only on first core

*Found in: Component config > FreeRTOS*

This version of FreeRTOS normally takes control of all cores of the CPU. Select this if you only want to start it on the first core. This is needed when e.g. another process needs complete control over the second core.

### FREERTOS\_CORETIMER

Xtensa timer to use as the FreeRTOS tick source

*Found in: Component config > FreeRTOS*

FreeRTOS needs a timer with an associated interrupt to use as the main tick source to increase counters, run timers and do pre-emptive multitasking with. There are multiple timers available to do this, with different interrupt priorities. Check

**Available options:**

- FREERTOS\_CORETIMER\_0
- FREERTOS\_CORETIMER\_1

## FREERTOS\_HZ

Tick rate (Hz)

*Found in: Component config > FreeRTOS*

Select the tick rate at which FreeRTOS does pre-emptive context switching.

## FREERTOS\_ASSERT\_ON\_UNTESTED\_FUNCTION

Halt when an SMP-untested function is called

*Found in: Component config > FreeRTOS*

Some functions in FreeRTOS have not been thoroughly tested yet when moving to the SMP implementation of FreeRTOS. When this option is enabled, these functions will throw an assert().

## FREERTOS\_CHECK\_STACKOVERFLOW

Check for stack overflow

*Found in: Component config > FreeRTOS*

FreeRTOS can check for stack overflows in threads and trigger an user function called `vApplicationStackOverflowHook` when this happens.

### Available options:

- FREERTOS\_CHECK\_STACKOVERFLOW\_NONE
- FREERTOS\_CHECK\_STACKOVERFLOW\_PTRVAL
- FREERTOS\_CHECK\_STACKOVERFLOW\_CANARY

## FREERTOS\_WATCHPOINT\_END\_OF\_STACK

Set a debug watchpoint as a stack overflow check

*Found in: Component config > FreeRTOS*

FreeRTOS can check if a stack has overflown its bounds by checking either the value of the stack pointer or by checking the integrity of canary bytes. (See `FREERTOS_CHECK_STACKOVERFLOW` for more information.) These checks only happen on a context switch, and the situation that caused the stack overflow may already be long gone by then. This option will use the debug memory watchpoint 1 (the second one) to allow breaking into the debugger (or panic'ing) as soon as any of the last 32 bytes of the stack of a task are overwritten. The side effect is that using gdb, you effectively only have one watchpoint; the 2nd one is overwritten as soon as a task switch happens.

This check only triggers if the stack overflow writes within 4 bytes of the end of the stack, rather than overshooting further, so it is worth combining this approach with one of the other stack overflow check methods.

When this watchpoint is hit, gdb will stop with a SIGTRAP message. When no JTAG OCD is attached, esp-idf will panic on an unhandled debug exception.

## FREERTOS\_INTERRUPT\_BACKTRACE

Enable backtrace from interrupt to task context

*Found in: Component config > FreeRTOS*

If this option is enabled, interrupt stack frame will be modified to point to the code of the interrupted task as its return address. This helps the debugger (or the panic handler) show a backtrace from the interrupt to the task which was interrupted. This also works for nested interrupts: higher level interrupt stack can be traced back to the lower level interrupt. This option adds 4 instructions to the interrupt dispatching code.

## FREERTOS\_THREAD\_LOCAL\_STORAGE\_POINTERS

Number of thread local storage pointers

*Found in: Component config > FreeRTOS*

FreeRTOS has the ability to store per-thread pointers in the task control block. This controls the number of pointers available.

This value must be at least 1. Index 0 is reserved for use by the pthreads API thread-local-storage. Other indexes can be used for any desired purpose.

## FREERTOS\_ASSERT

FreeRTOS assertions

*Found in: Component config > FreeRTOS*

Failed FreeRTOS configASSERT() assertions can be configured to behave in different ways.

**Available options:**

- FREERTOS\_ASSERT\_FAIL\_ABORT
- FREERTOS\_ASSERT\_FAIL\_PRINT\_CONTINUE
- FREERTOS\_ASSERT\_DISABLE

## FREERTOS\_IDLE\_TASK\_STACKSIZE

Idle Task stack size

*Found in: Component config > FreeRTOS*

The idle task has its own stack, sized in bytes. The default size is enough for most uses. Size can be reduced to 768 bytes if no (or simple) FreeRTOS idle hooks are used. The stack size may need to be increased above the default if the app installs idle hooks that use a lot of stack memory.

## FREERTOS\_ISR\_STACKSIZE

ISR stack size

*Found in: Component config > FreeRTOS*

The interrupt handlers have their own stack. The size of the stack can be defined here. Each processor has its own stack, so the total size occupied will be twice this.

## FREERTOS\_LEGACY\_HOOKS

Use FreeRTOS legacy hooks

*Found in: Component config > FreeRTOS*

FreeRTOS offers a number of hooks/callback functions that are called when a timer tick happens, the idle thread runs etc. esp-idf replaces these by runtime registerable hooks using the esp\_register\_freertos\_XXX\_hook system, but for legacy reasons the old hooks can also still be enabled. Please enable this only if you have code that for some reason can't be migrated to the esp\_register\_freertos\_XXX\_hook system.

## FREERTOS\_LEGACY\_IDLE\_HOOK

Enable legacy idle hook

*Found in: Component config > FreeRTOS*

If enabled, FreeRTOS will call a function called vApplicationIdleHook when the idle thread on a CPU is running. Please make sure your code defines such a function.

## FREERTOS\_LEGACY\_TICK\_HOOK

Enable legacy tick hook

*Found in: Component config > FreeRTOS*

If enabled, FreeRTOS will call a function called vApplicationTickHook when a FreeRTOS tick is executed. Please make sure your code defines such a function.

## FREERTOS\_MAX\_TASK\_NAME\_LEN

Maximum task name length

*Found in: Component config > FreeRTOS*

Changes the maximum task name length. Each task allocated will include this many bytes for a task name. Using a shorter value saves a small amount of RAM, a longer value allows more complex names.

For most uses, the default of 16 is OK.

## SUPPORT\_STATIC\_ALLOCATION

Enable FreeRTOS static allocation API

*Found in: Component config > FreeRTOS*

FreeRTOS gives the application writer the ability to instead provide the memory themselves, allowing the following objects to optionally be created without any memory being allocated dynamically:

- Tasks
- Software Timers (Daemon task is still dynamic. See documentation)
- Queues
- Event Groups

- Binary Semaphores
- Counting Semaphores
- Recursive Semaphores
- Mutexes

Whether it is preferable to use static or dynamic memory allocation is dependent on the application, and the preference of the application writer. Both methods have pros and cons, and both methods can be used within the same RTOS application.

Creating RTOS objects using statically allocated RAM has the benefit of providing the application writer with more control: RTOS objects can be placed at specific memory locations. The maximum RAM footprint can be determined at link time, rather than run time. The application writer does not need to concern themselves with graceful handling of memory allocation failures. It allows the RTOS to be used in applications that simply don't allow any dynamic memory allocation (although FreeRTOS includes allocation schemes that can overcome most objections).

## ENABLE\_STATIC\_TASK\_CLEAN\_UP\_HOOK

Enable static task clean up hook

*Found in: Component config > FreeRTOS*

Enable this option to make FreeRTOS call the static task clean up hook when a task is deleted.

Bear in mind that if this option is enabled you will need to implement the following function:

```
void vPortCleanUpTCB ( void *pxTCB ) {  
    // place clean up code here  
}
```

## TIMER\_TASK\_PRIORITY

FreeRTOS timer task priority

*Found in: Component config > FreeRTOS*

The timer service task (primarily) makes use of existing FreeRTOS features, allowing timer functionality to be added to an application with minimal impact on the size of the application's executable binary.

Use this constant to define the priority that the timer task will run at.

## TIMER\_TASK\_STACK\_DEPTH

FreeRTOS timer task stack size

*Found in: Component config > FreeRTOS*

The timer service task (primarily) makes use of existing FreeRTOS features, allowing timer functionality to be added to an application with minimal impact on the size of the application's executable binary.

Use this constant to define the size (in bytes) of the stack allocated for the timer task.

## TIMER\_QUEUE\_LENGTH

FreeRTOS timer queue length

*Found in: Component config > FreeRTOS*

FreeRTOS provides a set of timer related API functions. Many of these functions use a standard FreeRTOS queue to send commands to the timer service task. The queue used for this purpose is called the 'timer command queue'. The 'timer command queue' is private to the FreeRTOS timer implementation, and cannot be accessed directly.

For most uses the default value of 10 is OK.

## FREERTOS\_QUEUE\_REGISTRY\_SIZE

FreeRTOS queue registry size

*Found in: Component config > FreeRTOS*

FreeRTOS uses the queue registry as a means for kernel aware debuggers to locate queues, semaphores, and mutexes. The registry allows for a textual name to be associated with a queue for easy identification within a debugging GUI. A value of 0 will disable queue registry functionality, and a value larger than 0 will specify the number of queues/semaphores/mutexes that the registry can hold.

## FREERTOS\_USE\_TRACE\_FACILITY

Enable FreeRTOS trace facility

*Found in: Component config > FreeRTOS*

If enabled, configUSE\_TRACE\_FACILITY will be defined as 1 in FreeRTOS. This will allow the usage of trace facility functions such as uxTaskGetSystemState().

## FREERTOS\_USE\_STATS\_FORMATTING\_FUNCTIONS

Enable FreeRTOS stats formatting functions

*Found in: Component config > FreeRTOS*

If enabled, configUSE\_STATS\_FORMATTING\_FUNCTIONS will be defined as 1 in FreeRTOS. This will allow the usage of stats formatting functions such as vTaskList().

## FREERTOS\_GENERATE\_RUN\_TIME\_STATS

Enable FreeRTOS to collect run time stats

*Found in: Component config > FreeRTOS*

If enabled, configGENERATE\_RUN\_TIME\_STATS will be defined as 1 in FreeRTOS. This will allow FreeRTOS to collect information regarding the usage of processor time amongst FreeRTOS tasks. Run time stats are generated using either the ESP Timer or the CPU Clock as the clock source (Note that run time stats are only valid until the clock source overflows). The function vTaskGetRunTimeStats() will also be available if FREERTOS\_USE\_STATS\_FORMATTING\_FUNCTIONS and FREERTOS\_USE\_TRACE\_FACILITY are enabled. vTaskGetRunTimeStats() will display the run time of each task as a % of the total run time of all CPUs (task run time / no of CPUs) / (total run time / 100 )

## **FREERTOS\_RUN\_TIME\_STATS\_CLK**

Choose the clock source for run time stats

*Found in: Component config > FreeRTOS*

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:**

- FREERTOS\_RUN\_TIME\_STATS\_USING\_ESP\_TIMER
- FREERTOS\_RUN\_TIME\_STATS\_USING\_CPU\_CLK

## **FREERTOS\_DEBUG\_INTERNALS**

Debug FreeRTOS internals

*Found in: Component config > FreeRTOS*

Enable this option to show the menu with internal FreeRTOS debugging features. This option does not change any code by itself, it just shows/hides some options.

## **FREERTOS\_PORTMUX\_DEBUG**

Debug portMUX portENTER\_CRITICAL/portEXIT\_CRITICAL

*Found in: Component config > FreeRTOS*

If enabled, debug information (including integrity checks) will be printed to UART for the port-specific MUX implementation.

## **FREERTOS\_PORTMUX\_DEBUG\_RECURSIVE**

Debug portMUX Recursion

*Found in: Component config > FreeRTOS*

If enabled, additional debug information will be printed for recursive portMUX usage.

## **ADC-Calibration**

### **ADC\_CAL\_EFUSE\_TP\_ENABLE**

Use Two Point Values

*Found in: Component config > ADC-Calibration*

Some ESP32s have Two Point calibration values burned into eFuse BLOCK3. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using Two Point values if they are available.

## ADC\_CAL\_EFUSE\_VREF\_ENABLE

Use eFuse Vref

*Found in: Component config > ADC-Calibration*

Some ESP32s have Vref burned into eFuse BLOCK0. This option will allow the ADC calibration component to characterize the ADC-Voltage curve using eFuse Vref if it is available.

## ADC\_CAL\_LUT\_ENABLE

Use Lookup Tables

*Found in: Component config > ADC-Calibration*

This option will allow the ADC calibration component to use Lookup Tables to correct for non-linear behavior in 11db attenuation. Other attenuations do not exhibit non-linear behavior hence will not be affected by this option.

## Wear Levelling

### 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:**

- WL\_SECTOR\_SIZE\_512
- WL\_SECTOR\_SIZE\_4096

### 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:**

- WL\_SECTOR\_MODE\_PERF
- WL\_SECTOR\_MODE\_SAFE

## Heap memory debugging

### 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:**

- HEAP\_POISONING\_DISABLED
- HEAP\_POISONING\_LIGHT
- HEAP\_POISONING\_COMPREHENSIVE

### HEAP\_TRACING

Enable heap tracing

*Found in: Component config > Heap memory debugging*

Enables the heap tracing API defined in esp\_heap\_trace.h.

This function causes a moderate increase in IRAM code size and a minor increase in heap function (malloc/free/realloc) CPU overhead, even when the tracing feature is not used. So it's best to keep it disabled unless tracing is being used.

### 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.

## Partition Table

### **PARTITION\_TABLE\_TYPE**

Partition Table

*Found in: Partition Table*

The partition table to flash to the ESP32. The partition table determines where apps, data and other resources are expected to be found.

The predefined partition table CSV descriptions can be found in the components/partition\_table directory. Otherwise it's possible to create a new custom partition CSV for your application.

#### **Available options:**

- PARTITION\_TABLE\_SINGLE\_APP
- PARTITION\_TABLE\_TWO\_OTA
- PARTITION\_TABLE\_CUSTOM

### **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.

### **PARTITION\_TABLE\_CUSTOM\_APP\_BIN\_OFFSET**

Factory app partition offset

*Found in: Partition Table*

If using a custom partition table, specify the offset in the flash where 'make flash' should write the built app.

### **PARTITION\_TABLE\_CUSTOM\_PHY\_DATA\_OFFSET**

PHY data partition offset

*Found in: Partition Table*

If using a custom partition table, specify the offset in the flash where 'make flash' should write the initial PHY data file.

## **ESP32-specific**

### **ESP32\_DEFAULT\_CPU\_FREQ\_MHZ**

CPU frequency

*Found in: Component config > ESP32-specific*

CPU frequency to be set on application startup.

**Available options:**

- ESP32\_DEFAULT\_CPU\_FREQ\_80
- ESP32\_DEFAULT\_CPU\_FREQ\_160
- ESP32\_DEFAULT\_CPU\_FREQ\_240

## MEMMAP\_SMP

Reserve memory for two cores

*Found in: Component config > ESP32-specific*

The ESP32 contains two cores. If you plan to only use one, you can disable this item to save some memory. (ToDo: Make this automatically depend on uncore support)

## SPIRAM\_SUPPORT

Support for external, SPI-connected RAM

*Found in: Component config > ESP32-specific*

This enables support for an external SPI RAM chip, connected in parallel with the main SPI flash chip.

## SPI RAM config

### SPIRAM\_BOOT\_INIT

Initialize SPI RAM when booting the ESP32

*Found in: Component config > ESP32-specific > 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.

### SPIRAM\_USE

SPI RAM access method

*Found in: Component config > ESP32-specific > SPI RAM config*

The SPI RAM can be accessed in multiple methods: by just having it available as an unmanaged memory region in the ESP32 memory map, by integrating it in the ESP32s 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:**

- SPIRAM\_USE\_MEMMAP
- SPIRAM\_USE\_CAPS\_ALLOC
- SPIRAM\_USE\_MALLOC

## SPIRAM\_TYPE

Type of SPI RAM chip in use

*Found in: Component config > ESP32-specific > SPI RAM config*

### Available options:

- SPIRAM\_TYPE\_ESPPSRAM32

## SPIRAM\_SPEED

Set RAM clock speed

*Found in: Component config > ESP32-specific > SPI RAM config*

Select the speed for the SPI RAM chip. If SPI RAM is enabled, we only support three combinations of SPI speed mode we supported now:

1. Flash SPI running at 40Mhz and RAM SPI running at 40Mhz
2. Flash SPI running at 80Mhz and RAM SPI running at 40Mhz
3. Flash SPI running at 80Mhz and RAM SPI running at 80Mhz

**Note: If the third mode(80Mhz+80Mhz) is enabled, the VSPI port will be occupied by the system.**

Application code should never touch VSPI hardware in this case. The option to select 80MHz will only be visible if the flash SPI speed is also 80MHz. (ESPTOOLPY\_FLASHFREQ\_80M is true)

### Available options:

- SPIRAM\_SPEED\_40M
- SPIRAM\_SPEED\_80M

## SPIRAM\_MEMTEST

Run memory test on SPI RAM initialization

*Found in: Component config > ESP32-specific > SPI RAM config*

Runs a rudimentary memory test on initialization. Aborts when memory test fails. Disable this for slightly faster startup.

## SPIRAM\_CACHE\_WORKAROUND

Enable workaround for bug in SPI RAM cache for Rev1 ESP32s

*Found in: Component config > ESP32-specific > SPI RAM config*

Revision 1 of the ESP32 has a bug that can cause a write to PSRAM not to take place in some situations when the cache line needs to be fetched from external RAM and an interrupt occurs. This enables a fix in the compiler that makes sure the specific code that is vulnerable to this will not be emitted.

This will also not use any bits of newlib that are located in ROM, opting for a version that is compiled with the workaround and located in flash instead.

## SPIRAM\_MALLOC\_ALWAYSINTERNAL

Maximum malloc() size, in bytes, to always put in internal memory

*Found in: Component config > ESP32-specific > 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.

## WIFI\_LWIP\_ALLOCATION\_FROM\_SPIRAM\_FIRST

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, allocate internal memory

*Found in: Component config > ESP32-specific > SPI RAM config*

Try to allocate memories of WiFi and LWIP in SPIRAM firstly. If failed, try to allocate internal memory then.

## SPIRAM\_MALLOC\_RESERVE\_INTERNAL

Reserve this amount of bytes for data that specifically needs to be in DMA or internal memory

*Found in: Component config > ESP32-specific > 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.

## SPIRAM\_ALLOW\_STACK\_EXTERNAL\_MEMORY

Allow external memory as an argument to xTaskCreateStatic

*Found in: Component config > ESP32-specific > SPI RAM config*

Because some bits of the ESP32 code environment cannot be recompiled with the cache workaround, normally tasks cannot be safely run with their stack residing in external memory; for this reason xTaskCreate and friends always allocate stack in internal memory and xTaskCreateStatic will check if the memory passed to it is in internal memory. If you have a task that needs a large amount of stack and does not call on ROM code in any way (no direct calls, but also no Bluetooth/WiFi), you can try to disable this and use xTaskCreateStatic to create the tasks stack in external memory.

## ESP32\_TRAX

Use TRAX tracing feature

*Found in: Component config > ESP32-specific*

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

### ESP32\_TRAX\_TWOBANKS

Reserve memory for tracing both pro as well as app cpu execution

*Found in: Component config > ESP32-specific*

The ESP32 contains a feature which allows you to trace the execution path the processor has taken through the program. This is stored in a chunk of 32K (16K for single-processor) of memory that can't be used for general purposes anymore. Disable this if you do not know what this is.

### ESP32\_COREDUMP\_TO\_FLASH\_OR\_UART

Core dump destination

*Found in: Component config > ESP32-specific*

Select place to store core dump: flash, uart or none (to disable core dumps generation).

If core dump is configured to be stored in flash and custom partition table is used add corresponding entry to your CSV. For examples, please see predefined partition table CSV descriptions in the components/partition\_table directory.

#### Available options:

- ESP32\_ENABLE\_COREDUMP\_TO\_FLASH
- ESP32\_ENABLE\_COREDUMP\_TO\_UART
- ESP32\_ENABLE\_COREDUMP\_TO\_NONE

### ESP32\_CORE\_DUMP\_UART\_DELAY

Core dump print to UART delay

*Found in: Component config > ESP32-specific*

Config delay (in ms) before printing core dump to UART. Delay can be interrupted by pressing Enter key.

### ESP32\_CORE\_DUMP\_LOG\_LEVEL

Core dump module logging level

*Found in: Component config > ESP32-specific*

Config core dump module logging level (0-5).

### NUMBER\_OF\_UNIVERSAL\_MAC\_ADDRESS

Number of universally administered (by IEEE) MAC address

*Found in: Component config > ESP32-specific*

Configure the number of universally administered (by IEEE) MAC addresses. During initialisation, MAC addresses for each network interface are generated or derived from a single base MAC address. If the number of universal MAC addresses is four, all four interfaces (WiFi station, WiFi softap, Bluetooth and Ethernet) receive a universally administered MAC address. These are generated sequentially by adding 0, 1, 2 and 3 (respectively) to the final octet of the base MAC address. If the number of universal MAC addresses is two, only two interfaces (WiFi station and Bluetooth) receive a universally administered MAC address. These are generated sequentially by adding 0 and 1 (respectively) to the base MAC address. The remaining two interfaces (WiFi softap and Ethernet) receive local MAC addresses. These are derived from the universal WiFi station and Bluetooth MAC addresses, respectively. When using the default (Espressif-assigned) base MAC address, either setting can be used. When using a custom universal MAC address range, the correct setting will depend on the allocation of MAC addresses in this range (either 2 or 4 per device.)

**Available options:**

- TWO\_UNIVERSAL\_MAC\_ADDRESS
- FOUR\_UNIVERSAL\_MAC\_ADDRESS

## SYSTEM\_EVENT\_QUEUE\_SIZE

System event queue size

*Found in: Component config > ESP32-specific*

Config system event queue size in different application.

## SYSTEM\_EVENT\_TASK\_STACK\_SIZE

Event loop task stack size

*Found in: Component config > ESP32-specific*

Config system event task stack size in different application.

## MAIN\_TASK\_STACK\_SIZE

Main task stack size

*Found in: Component config > ESP32-specific*

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.

## IPC\_TASK\_STACK\_SIZE

Inter-Processor Call (IPC) task stack size

*Found in: Component config > ESP32-specific*

Configure the IPC tasks stack size. One IPC task runs on each core (in dual core mode), and allows for cross-core function calls.

See IPC documentation for more details.

The default stack size should be enough for most common use cases. It can be shrunk if you are sure that you do not use any custom IPC functionality.

## TIMER\_TASK\_STACK\_SIZE

High-resolution timer task stack size

*Found in: Component config > ESP32-specific*

Configure the stack size of esp\_timer/ets\_timer task. This task is used to dispatch callbacks of timers created using ets\_timer and esp\_timer APIs. If you are seeing stack overflow errors in timer task, increase this value.

Note that this is not the same as FreeRTOS timer task. To configure FreeRTOS timer task size, see “FreeRTOS timer task stack size” option in “FreeRTOS” menu.

## NEWLIB\_STDOUT\_LINE\_ENDING

Line ending for UART output

*Found in: Component config > ESP32-specific*

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:**

- NEWLIB\_STDOUT\_LINE\_ENDING\_CRLF
- NEWLIB\_STDOUT\_LINE\_ENDING\_LF
- NEWLIB\_STDOUT\_LINE\_ENDING\_CR

## NEWLIB\_STDIN\_LINE\_ENDING

Line ending for UART input

*Found in: Component config > ESP32-specific*

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:**

- NEWLIB\_STDIN\_LINE\_ENDING\_CRLF
- NEWLIB\_STDIN\_LINE\_ENDING\_LF
- NEWLIB\_STDIN\_LINE\_ENDING\_CR

## NEWLIB\_NANO\_FORMAT

Enable ‘nano’ formatting options for printf/scanf family

*Found in: Component config > ESP32-specific*

ESP32 ROM contains parts of newlib C library, including printf/scanf family of functions. These functions have been compiled with so-called “nano” formatting option. This option doesn’t support 64-bit integer formats and C99 features, such as positional arguments.

For more details about “nano” formatting option, please see newlib readme file, search for ‘–enable-newlib-nano-formatted-io’: <https://sourceware.org/newlib/README>

If this option is enabled, build system will use functions available in ROM, reducing the application binary size. Functions available in ROM run faster than functions which run from flash. Functions available in ROM can also run when flash instruction cache is disabled.

If you need 64-bit integer formatting support or C99 features, keep this option disabled.

## CONSOLE\_UART

UART for console output

*Found in: Component config > ESP32-specific*

Select whether to use UART for console output (through stdout and stderr).

- Default is to use UART0 on pins GPIO1(TX) and GPIO3(RX).
- If “Custom” is selected, UART0 or UART1 can be chosen, and any pins can be selected.
- If “None” is selected, there will be no console output on any UART, except for initial output from ROM bootloader. This output can be further suppressed by bootstrapping GPIO13 pin to low logic level.

### Available options:

- CONSOLE\_UART\_DEFAULT
- CONSOLE\_UART\_CUSTOM
- CONSOLE\_UART\_NONE

## CONSOLE\_UART\_NUM

UART peripheral to use for console output (0-1)

*Found in: Component config > ESP32-specific*

Due of a ROM bug, UART2 is not supported for console output via ets\_printf.

### Available options:

- CONSOLE\_UART\_CUSTOM\_NUM\_0
- CONSOLE\_UART\_CUSTOM\_NUM\_1

## CONSOLE\_UART\_TX\_GPIO

UART TX on GPIO#

*Found in: Component config > ESP32-specific*

## CONSOLE\_UART\_RX\_GPIO

UART RX on GPIO#

*Found in: Component config > ESP32-specific*

## CONSOLE\_UART\_BAUDRATE

UART console baud rate

*Found in: Component config > ESP32-specific*

## ULP\_COPROC\_ENABLED

Enable Ultra Low Power (ULP) Coprocessor

*Found in: Component config > ESP32-specific*

Set to 'y' if you plan to load a firmware for the coprocessor.

If this option is enabled, further coprocessor configuration will appear in the Components menu.

## ULP\_COPROC\_RESERVE\_MEM

RTC slow memory reserved for coprocessor

*Found in: Component config > ESP32-specific*

Bytes of memory to reserve for ULP coprocessor firmware & data.

Data is reserved at the beginning of RTC slow memory.

## ESP32\_PANIC

Panic handler behaviour

*Found in: Component config > ESP32-specific*

If FreeRTOS detects unexpected behaviour or an unhandled exception, the panic handler is invoked. Configure the panic handlers action here.

**Available options:**

- ESP32\_PANIC\_PRINT\_HALT
- ESP32\_PANIC\_PRINT\_REBOOT
- ESP32\_PANIC\_SILENT\_REBOOT
- ESP32\_PANIC\_GDBSTUB

## ESP32\_DEBUG\_OCDAWARE

Make exception and panic handlers JTAG/OCD aware

*Found in: Component config > ESP32-specific*

The FreeRTOS panic and unhandled exception handlers can detect a JTAG OCD debugger and instead of panicking, have the debugger stop on the offending instruction.

## INT\_WDT

Interrupt watchdog

*Found in: Component config > ESP32-specific*

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.

## INT\_WDT\_TIMEOUT\_MS

Interrupt watchdog timeout (ms)

*Found in: Component config > ESP32-specific*

The timeout of the watchdog, in milliseconds. Make this higher than the FreeRTOS tick rate.

## INT\_WDT\_CHECK\_CPU1

Also watch CPU1 tick interrupt

*Found in: Component config > ESP32-specific*

Also detect if interrupts on CPU 1 are disabled for too long.

## TASK\_WDT

Initialize Task Watchdog Timer on startup

*Found in: Component config > ESP32-specific*

The Task Watchdog Timer can be used to make sure individual tasks are still running. Enabling this option will cause the Task Watchdog Timer to be initialized automatically at startup. The Task Watchdog timer can be initialized after startup as well (see Task Watchdog Timer API Reference)

## TASK\_WDT\_PANIC

Invoke panic handler on Task Watchdog timeout

*Found in: Component config > ESP32-specific*

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)

## TASK\_WDT\_TIMEOUT\_S

Task Watchdog timeout period (seconds)

*Found in: Component config > ESP32-specific*

Timeout period configuration for the Task Watchdog Timer in seconds. This is also configurable at run time (see Task Watchdog Timer API Reference)

## TASK\_WDT\_CHECK\_IDLE\_TASK\_CPU0

Watch CPU0 Idle Task

*Found in: Component config > ESP32-specific*

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.

## TASK\_WDT\_CHECK\_IDLE\_TASK\_CPU1

Watch CPU1 Idle Task

*Found in: Component config > ESP32-specific*

If this option is enabled, the Task Watchdog Timer will watch the CPU1 Idle Task.

## BROWNOUT\_DET

Hardware brownout detect & reset

*Found in: Component config > ESP32-specific*

The ESP32 has a built-in brownout detector which can detect if the voltage is lower than a specific value. If this happens, it will reset the chip in order to prevent unintended behaviour.

## BROWNOUT\_DET\_LVL\_SEL

Brownout voltage level

*Found in: Component config > ESP32-specific*

The brownout detector will reset the chip when the supply voltage is approximately below this level. Note that there may be some variation of brownout voltage level between each ESP32 chip.

**Available options:**

- BROWNOUT\_DET\_LVL\_SEL\_0
- BROWNOUT\_DET\_LVL\_SEL\_1
- BROWNOUT\_DET\_LVL\_SEL\_2
- BROWNOUT\_DET\_LVL\_SEL\_3
- BROWNOUT\_DET\_LVL\_SEL\_4

- BROWNOUT\_DET\_LVL\_SEL\_5
- BROWNOUT\_DET\_LVL\_SEL\_6
- BROWNOUT\_DET\_LVL\_SEL\_7

## ESP32\_TIME\_SYSCALL

Timers used for gettimeofday function

*Found in: Component config > ESP32-specific*

This setting defines which hardware timers are used to implement ‘gettimeofday’ and ‘time’ functions in C library.

- If both high-resolution and RTC timers are used, timekeeping will continue in deep sleep. Time will be reported at 1 microsecond resolution. This is the default, and the recommended option.
- If only high-resolution timer is used, gettimeofday will provide time at microsecond resolution. Time will not be preserved when going into deep sleep mode.
- If only RTC timer is used, timekeeping will continue in deep sleep, but time will be measured at 6.(6) microsecond resolution. Also the gettimeofday function itself may take longer to run.
- If no timers are used, gettimeofday and time functions return -1 and set errno to ENOSYS.
- When RTC is used for timekeeping, two RTC\_STORE registers are used to keep time in deep sleep mode.

### Available options:

- ESP32\_TIME\_SYSCALL\_USE\_RTC\_FRC1
- ESP32\_TIME\_SYSCALL\_USE\_RTC
- ESP32\_TIME\_SYSCALL\_USE\_FRC1
- ESP32\_TIME\_SYSCALL\_USE\_NONE

## ESP32\_RTC\_CLOCK\_SOURCE

RTC clock source

*Found in: Component config > ESP32-specific*

Choose which clock is used as RTC clock source.

### Available options:

- ESP32\_RTC\_CLOCK\_SOURCE\_INTERNAL\_RC
- ESP32\_RTC\_CLOCK\_SOURCE\_EXTERNAL\_CRYSTAL

## ESP32\_RTC\_CLK\_CAL\_CYCLES

Number of cycles for RTC\_SLOW\_CLK calibration

*Found in: Component config > ESP32-specific*

When the startup code initializes RTC\_SLOW\_CLK, it can perform calibration by comparing the RTC\_SLOW\_CLK frequency with main XTAL frequency. This option sets the number of

RTC\_SLOW\_CLK cycles measured by the calibration routine. Higher numbers increase calibration precision, which may be important for applications which spend a lot of time in deep sleep. Lower numbers reduce startup time.

When this option is set to 0, clock calibration will not be performed at startup, and approximate clock frequencies will be assumed:

- 150000 Hz if internal RC oscillator is used as clock source
- 32768 Hz if the 32k crystal oscillator is used

## ESP32\_DEEP\_SLEEP\_WAKEUP\_DELAY

Extra delay in deep sleep wake stub (in us)

*Found in: Component config > ESP32-specific*

When ESP32 exits deep sleep, the CPU and the flash chip are powered on at the same time. CPU will run deep sleep stub first, and then proceed to load code from flash. Some flash chips need sufficient time to pass between power on and first read operation. By default, without any extra delay, this time is approximately 900us, although some flash chip types need more than that.

By default extra delay is set to 2000us. When optimizing startup time for applications which require it, this value may be reduced.

If you are seeing “flash read err, 1000” message printed to the console after deep sleep reset, try increasing this value.

## ESP32\_XTAL\_FREQ\_SEL

Main XTAL frequency

*Found in: Component config > ESP32-specific*

ESP32 currently supports the following XTAL frequencies:

- 26 MHz
- 40 MHz

Startup code can automatically estimate XTAL frequency. This feature uses the internal 8MHz oscillator as a reference. Because the internal oscillator frequency is temperature dependent, it is not recommended to use automatic XTAL frequency detection in applications which need to work at high ambient temperatures and use high-temperature qualified chips and modules.

**Available options:**

- ESP32\_XTAL\_FREQ\_40
- ESP32\_XTAL\_FREQ\_26
- ESP32\_XTAL\_FREQ\_AUTO

## DISABLE\_BASIC\_ROM\_CONSOLE

Permanently disable BASIC ROM Console

*Found in: Component config > ESP32-specific*

If set, the first time the app boots it will disable the BASIC ROM Console permanently (by burning an efuse).

Otherwise, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

(Enabling secure boot also disables the BASIC ROM Console by default.)

## NO\_BLOBS

No Binary Blobs

*Found in: Component config > ESP32-specific*

If enabled, this disables the linking of binary libraries in the application build. Note that after enabling this Wi-Fi/Bluetooth will not work.

## ESP\_TIMER\_PROFILING

Enable esp\_timer profiling features

*Found in: Component config > ESP32-specific*

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.

## COMPATIBLE\_PRE\_V2\_1\_BOOTLOADERS

App compatible with bootloaders before IDF v2.1

*Found in: Component config > ESP32-specific*

Bootloaders before IDF v2.1 did less initialisation of the system clock. This setting needs to be enabled to build an app which can be booted by these older bootloaders.

If this setting is enabled, the app can be booted by any bootloader from IDF v1.0 up to the current version.

If this setting is disabled, the app can only be booted by bootloaders from IDF v2.1 or newer.

Enabling this setting adds approximately 1KB to the app's IRAM usage.

## Wi-Fi

### SW\_COEXIST\_ENABLE

Software controls WiFi/Bluetooth coexistence

*Found in: Component config > Wi-Fi*

If enabled, WiFi & Bluetooth coexistence is controlled by software rather than hardware. Recommended for heavy traffic scenarios. Both coexistence configuration options are automatically managed, no user intervention is required.

## ESP32\_WIFI\_STATIC\_RX\_BUFFER\_NUM

Max number of WiFi static RX buffers

*Found in: Component config > Wi-Fi*

Set the number of WiFi static RX buffers. Each buffer takes approximately 1.6KB of RAM. The static rx buffers are allocated when esp\_wifi\_init is called, they are not freed until esp\_wifi\_deinit is called.

WiFi hardware use these buffers to receive all 802.11 frames. A higher number may allow higher throughput but increases memory use. If ESP32\_WIFI\_AMPDU\_RX\_ENABLED is enabled, this value is recommended to set equal or bigger than ESP32\_WIFI\_RX\_BA\_WIN in order to achieve better throughput and compatibility with both stations and APs.

## ESP32\_WIFI\_DYNAMIC\_RX\_BUFFER\_NUM

Max number of WiFi dynamic RX buffers

*Found in: Component config > Wi-Fi*

Set the number of WiFi dynamic RX buffers, 0 means unlimited RX buffers will be allocated (provided sufficient free RAM). The size of each dynamic RX buffer depends on the size of the received data frame.

For each received data frame, the WiFi driver makes a copy to an RX buffer and then delivers it to the high layer TCP/IP stack. The dynamic RX buffer is freed after the higher layer has successfully received the data frame.

For some applications, WiFi data frames may be received faster than the application can process them. In these cases we may run out of memory if RX buffer number is unlimited (0).

If a dynamic RX buffer limit is set, it should be at least the number of static RX buffers.

## ESP32\_WIFI\_TX\_BUFFER

Type of WiFi TX buffers

*Found in: Component config > Wi-Fi*

Select type of WiFi TX buffers:

If “Static” is selected, WiFi TX buffers are allocated when WiFi is initialized and released when WiFi is de-initialized. The size of each static TX buffer is fixed to about 1.6KB.

If “Dynamic” is selected, each WiFi TX buffer is allocated as needed when a data frame is delivered to the Wifi driver from the TCP/IP stack. The buffer is freed after the data frame has been sent by the WiFi driver. The size of each dynamic TX buffer depends on the length of each data frame sent by the TCP/IP layer.

If PSRAM is enabled, “Static” should be selected to guarantee enough WiFi TX buffers. If PSRAM is disabled, “Dynamic” should be selected to improve the utilization of RAM.

**Available options:**

- ESP32\_WIFI\_STATIC\_TX\_BUFFER
- ESP32\_WIFI\_DYNAMIC\_TX\_BUFFER

## ESP32\_WIFI\_STATIC\_TX\_BUFFER\_NUM

Max number of WiFi static TX buffers

*Found in: Component config > Wi-Fi*

Set the number of WiFi static TX buffers. Each buffer takes approximately 1.6KB of RAM. The static RX buffers are allocated when `esp_wifi_init()` is called, they are not released until `esp_wifi_deinit()` is called.

For each transmitted data frame from the higher layer TCP/IP stack, the WiFi driver makes a copy of it in a TX buffer. For some applications especially UDP applications, the upper layer can deliver frames faster than WiFi layer can transmit. In these cases, we may run out of TX buffers.

## ESP32\_WIFI\_DYNAMIC\_TX\_BUFFER\_NUM

Max number of WiFi dynamic TX buffers

*Found in: Component config > Wi-Fi*

Set the number of WiFi dynamic TX buffers. The size of each dynamic TX buffer is not fixed, it depends on the size of each transmitted data frame.

For each transmitted frame from the higher layer TCP/IP stack, the WiFi driver makes a copy of it in a TX buffer. For some applications, especially UDP applications, the upper layer can deliver frames faster than WiFi layer can transmit. In these cases, we may run out of TX buffers.

## ESP32\_WIFI\_AMPDU\_TX\_ENABLED

WiFi AMPDU TX

*Found in: Component config > Wi-Fi*

Select this option to enable AMPDU TX feature

## ESP32\_WIFI\_TX\_BA\_WIN

WiFi AMPDU TX BA window size

*Found in: Component config > Wi-Fi*

Set the size of WiFi Block Ack TX window. Generally a bigger value means higher throughput but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP TX throughput with iperf etc. For iperf test in shieldbox, the recommended value is 9~12.

## ESP32\_WIFI\_AMPDU\_RX\_ENABLED

WiFi AMPDU RX

*Found in: Component config > Wi-Fi*

Select this option to enable AMPDU RX feature

## ESP32\_WIFI\_RX\_BA\_WIN

WiFi AMPDU RX BA window size

*Found in: Component config > Wi-Fi*

Set the size of WiFi Block Ack RX window. Generally a bigger value means higher throughput and better compatibility but more memory. Most of time we should NOT change the default value unless special reason, e.g. test the maximum UDP RX throughput with iperf etc. For iperf test in shieldbox, the recommended value is 9~12. If PSRAM is used and WiFi memory is preferred to allocate in PSRAM first, the default and minimum value should be 16 to achieve better throughput and compatibility with both stations and APs.

## ESP32\_WIFI\_NVS\_ENABLED

WiFi NVS flash

*Found in: Component config > Wi-Fi*

Select this option to enable WiFi NVS flash

## PHY

### ESP32\_PHY\_CALIBRATION\_AND\_DATA\_STORAGE

Store phy calibration data in NVS

*Found in: Component config > PHY*

If this option is enabled, NVS will be initialized and calibration data will be loaded from there. PHY calibration will be skipped on deep sleep wakeup. If calibration data is not found, full calibration will be performed and stored in NVS. Normally, only partial calibration will be performed. If this option is disabled, full calibration will be performed.

If it's easy that your board calibrate bad data, choose 'n'. Two cases for example, you should choose 'n':  
1.If your board is easy to be booted up with antenna disconnected. 2.Because of your board design, each time when you do calibration, the result are too unstable. If unsure, choose 'y'.

### ESP32\_PHY\_INIT\_DATA\_IN\_PARTITION

Use a partition to store PHY init data

*Found in: Component config > PHY*

If enabled, PHY init data will be loaded from a partition. When using a custom partition table, make sure that PHY data partition is included (type: 'data', subtype: 'phy'). With default partition tables, this is done automatically. If PHY init data is stored in a partition, it has to be flashed there, otherwise runtime error will occur.

If this option is not enabled, PHY init data will be embedded into the application binary.

If unsure, choose 'n'.

## ESP32\_PHY\_MAX\_WIFI\_TX\_POWER

Max WiFi TX power (dBm)

*Found in: Component config > PHY*

Set maximum transmit power for WiFi radio. Actual transmit power for high data rates may be lower than this setting.

## Power Management

### 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.

### PM\_DFS\_INIT\_AUTO

Enable dynamic frequency scaling (DFS) at startup

*Found in: Component config > Power Management*

If enabled, startup code configures dynamic frequency scaling. Max CPU frequency is set to CONFIG\_ESP32\_DEFAULT\_CPU\_FREQ\_MHZ setting, min frequency is set to XTAL frequency. If disabled, DFS will not be active until the application configures it using esp\_pm\_configure function.

### PM\_USE\_RTC\_TIMER\_REF

Use RTC timer to prevent time drift (EXPERIMENTAL)

*Found in: Component config > Power Management*

When APB clock frequency changes, high-resolution timer (esp\_timer) scale and base value need to be adjusted. Each adjustment may cause small error, and over time such small errors may cause time drift. If this option is enabled, RTC timer will be used as a reference to compensate for the drift. It is recommended that this option is only used if 32k XTAL is selected as RTC clock source.

### PM\_PROFILING

Enable profiling counters for PM locks

*Found in: Component config > Power Management*

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.

## PM\_TRACE

Enable debug tracing of PM using GPIOs

*Found in: Component config > Power Management*

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.

## Log output

### LOG\_DEFAULT\_LEVEL

Default log verbosity

*Found in: Component config > Log output*

Specify how much output to see in logs by default. You can set lower verbosity level at runtime using `esp_log_level_set` function.

Note that this setting limits which log statements are compiled into the program. So setting this to, say, “Warning” would mean that changing log level to “Debug” at runtime will not be possible.

#### Available options:

- LOG\_DEFAULT\_LEVEL\_NONE
- LOG\_DEFAULT\_LEVEL\_ERROR
- LOG\_DEFAULT\_LEVEL\_WARN
- LOG\_DEFAULT\_LEVEL\_INFO
- LOG\_DEFAULT\_LEVEL\_DEBUG
- LOG\_DEFAULT\_LEVEL\_VERBOSE

### 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.

## PThreads

### ESP32\_PTHREAD\_TASK\_PRIO\_DEFAULT

Default task priority

*Found in: Component config > PThreads*

Priority used to create new tasks with default pthread parameters.

## ESP32\_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.

## Application Level Tracing

### ESP32\_APPTRACE\_DESTINATION

Data Destination

*Found in: Component config > Application Level Tracing*

Select destination for application trace: trace memory or none (to disable).

**Available options:**

- ESP32\_APPTRACE\_DEST\_TRAX
- ESP32\_APPTRACE\_DEST\_NONE

### ESP32\_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.

### ESP32\_APPTRACE\_POSTMORTEM\_FLUSH\_TRAX\_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.

### ESP32\_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

### SYSVIEW\_ENABLE

SystemView Tracing Enable

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables support for SEGGER SystemView tracing functionality.

### SYSVIEW\_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:

- SYSVIEW\_TS\_SOURCE\_CCOUNT
- SYSVIEW\_TS\_SOURCE\_TIMER\_00
- SYSVIEW\_TS\_SOURCE\_TIMER\_01
- SYSVIEW\_TS\_SOURCE\_TIMER\_10
- SYSVIEW\_TS\_SOURCE\_TIMER\_11
- SYSVIEW\_TS\_SOURCE\_ESP\_TIMER

### SYSVIEW\_EVT\_OVERFLOW\_ENABLE

Trace Buffer Overflow Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Trace Buffer Overflow” event.

### SYSVIEW\_EVT\_ISR\_ENTER\_ENABLE

ISR Enter Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “ISR Enter” event.

### SYSVIEW\_EVT\_ISR\_EXIT\_ENABLE

ISR Exit Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “ISR Exit” event.

## **SYSVIEW\_EVT\_ISR\_TO\_SCHEDULER\_ENABLE**

ISR Exit to Scheduler Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “ISR to Scheduler” event.

## **SYSVIEW\_EVT\_TASK\_START\_EXEC\_ENABLE**

Task Start Execution Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Task Start Execution” event.

## **SYSVIEW\_EVT\_TASK\_STOP\_EXEC\_ENABLE**

Task Stop Execution Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Task Stop Execution” event.

## **SYSVIEW\_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.

## **SYSVIEW\_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.

## **SYSVIEW\_EVT\_TASK\_CREATE\_ENABLE**

Task Create Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Task Create” event.

## **SYSVIEW\_EVT\_TASK\_TERMINATE\_ENABLE**

Task Terminate Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Task Terminate” event.

## **SYSVIEW\_EVT\_IDLE\_ENABLE**

System Idle Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “System Idle” event.

## **SYSVIEW\_EVT\_TIMER\_ENTER\_ENABLE**

Timer Enter Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Timer Enter” event.

## **SYSVIEW\_EVT\_TIMER\_EXIT\_ENABLE**

Timer Exit Event

*Found in: Component config > Application Level Tracing > FreeRTOS SystemView Tracing*

Enables “Timer Exit” event.

## **Ethernet**

### **DMA\_RX\_BUF\_NUM**

Number of DMA RX buffers

*Found in: Component config > Ethernet*

Number of DMA receive buffers. Each buffer is 1600 bytes. Buffers are allocated statically. Larger number of buffers increases throughput. If enable flow ctrl, the num must be above 9 .

### **DMA\_TX\_BUF\_NUM**

Number of DMA TX buffers

*Found in: Component config > Ethernet*

Number of DMA transmit buffers. Each buffer is 1600 bytes. Buffers are allocated statically. Larger number of buffers increases throughput.

### **EMAC\_L2\_TO\_L3\_RX\_BUF\_MODE**

Enable copy between Layer2 and Layer3

*Found in: Component config > Ethernet*

If this options is selected, a copy of each received buffer will be created when passing it from the Ethernet MAC (L2) to the IP stack (L3). Otherwise, IP stack will receive pointers to the DMA buffers used by Ethernet MAC.

When Ethernet MAC doesn't have any unused buffers left, it will drop incoming packets (flow control may help with this problem, to some extent).

The buffers for the IP stack are allocated from the heap, so the total number of receive buffers is limited by the available heap size, if this option is selected.

If unsure, choose n.

## EMAC\_TASK\_PRIORITY

EMAC\_TASK\_PRIORITY

*Found in: Component config > Ethernet*

Ethernet MAC task priority.

## Bootloader config

### LOG\_BOOTLOADER\_LEVEL

Bootloader log verbosity

*Found in: Bootloader config*

Specify how much output to see in bootloader logs.

#### Available options:

- LOG\_BOOTLOADER\_LEVEL\_NONE
- LOG\_BOOTLOADER\_LEVEL\_ERROR
- LOG\_BOOTLOADER\_LEVEL\_WARN
- LOG\_BOOTLOADER\_LEVEL\_INFO
- LOG\_BOOTLOADER\_LEVEL\_DEBUG
- LOG\_BOOTLOADER\_LEVEL\_VERBOSE

### BOOTLOADER\_SPI\_WP\_PIN

SPI Flash WP Pin when customising pins via efuse (read help)

*Found in: Bootloader config*

This value is ignored unless flash mode is set to QIO or QOUT *and* the SPI flash pins have been overridden by setting the efuses SPI\_PAD\_CONFIG\_XXX.

When this is the case, the Efuse config only defines 3 of the 4 Quad I/O data pins. The WP pin (aka ESP32 pin "SD\_DATA\_3" or SPI flash pin "IO2") is not specified in Efuse. That pin number is compiled into the bootloader instead.

The default value (GPIO 7) is correct for WP pin on ESP32-D2WD integrated flash.

## 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:

- BOOTLOADER\_VDDSDIO\_BOOST\_1\_8V
- BOOTLOADER\_VDDSDIO\_BOOST\_1\_9V

## Security features

### SECURE\_BOOT\_ENABLED

Enable 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.

Refer to <https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html> before enabling.

### SECURE\_BOOTLOADER\_MODE

Secure bootloader mode

*Found in: Security features*

### Available options:

- SECURE\_BOOTLOADER\_ONE\_TIME\_FLASH
- SECURE\_BOOTLOADER\_REFLASHABLE

### SECURE\_BOOT\_BUILD\_SIGNED\_BINARIES

Sign binaries during build

*Found in: Security features*

Once secure boot is enabled, bootloader will only boot if partition table and app image are signed.

If enabled, 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` (for example, on a remote signing server.)

## SECURE\_BOOT\_SIGNING\_KEY

Secure boot private signing key

*Found in: Security features*

Path to the key file used to sign partition tables and app images for secure boot. Once secure boot is enabled, bootloader will only boot if partition table and app image are signed.

Key file is an ECDSA private key (NIST256p curve) in PEM format.

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 `docs/security/secure-boot.rst` for details.

## 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. This key is compiled into the bootloader, and may also be used to verify signatures on OTA images after download.

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 <https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html> before enabling.

## 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 <https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html> before enabling.

## FLASH\_ENCRYPTION\_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 <https://esp-idf.readthedocs.io/en/latest/security/flash-encryption.html> before enabling.

## FLASH\_ENCRYPTION\_INSECURE

Allow potentially insecure options

*Found in: Security features*

You can disable some of the default protections offered by flash encryption, in order to enable testing or a custom combination of security features.

Only enable these options if you are very sure.

Refer to docs/security/secure-boot.rst and docs/security/flash-encryption.rst for details.

### Potentially insecure options

## SECURE\_BOOT\_ALLOW\_ROM\_BASIC

Leave ROM BASIC Interpreter available on reset

*Found in: Security features > Potentially insecure options*

By default, the BASIC ROM Console starts on reset if no valid bootloader is read from the flash.

When either flash encryption or secure boot are enabled, the default is to disable this BASIC fallback mode permanently via efuse.

If this option is set, this efuse is not burned and the BASIC ROM Console may remain accessible. Only set this option in testing environments.

## 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.

## FLASH\_ENCRYPTION\_UART\_BOOTLOADER\_ALLOW\_ENCRYPT

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.

## FLASH\_ENCRYPTION\_UART\_BOOTLOADER\_ALLOW\_DECRYPT

Leave UART bootloader decryption enabled

*Found in: Security features > Potentially insecure options*

If not set (default), the bootloader will permanently disable UART bootloader decryption access on first boot. If set, the UART bootloader will still be able to access hardware decryption.

Only set this option in testing environments. Setting this option allows complete bypass of flash encryption.

## FLASH\_ENCRYPTION\_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.

## SECURE\_BOOT\_TEST\_MODE

Secure boot test mode: don't permanently set any efuses

*Found in: Security features > Potentially insecure options*

If this option is set, all permanent secure boot changes (via Efuse) are disabled.

Log output will state changes which would be applied, but they will not be.

This option is for testing purposes only - it completely disables secure boot protection.

## FLASH\_ENCRYPTION\_DISABLE\_PLAINTEXT

Disable serial reflashing of plaintext firmware

*Found in: Security features*

If this option is enabled, flash encryption is permanently enabled after first boot by write-protecting the FLASH\_CRYPT\_CNT efuse. This is the recommended configuration for a secure production system.

If this option is disabled, FLASH\_CRYPT\_CNT is left writeable and up to 4 plaintext re-flashes are allowed. An attacker with physical access will be able to read out encrypted flash contents until all plaintext re-flashes have been used up.

If this option is disabled and hardware Secure Boot is enabled, Secure Boot must be configured in Re-flashable mode so that a new Secure Boot digest can be flashed at the same time as plaintext firmware. This combination is not secure and should not be used for a production system.

## OpenSSL

### OPENSSL\_DEBUG

Enable OpenSSL debugging

*Found in: Component config > OpenSSL*

Enable OpenSSL debugging function.

If the option is enabled, “SSL\_DEBUG” works.

## OPENSSSL\_DEBUG\_LEVEL

OpenSSL debugging level

*Found in: Component config > OpenSSL*

OpenSSL debugging level.

Only function whose debugging level is higher than “OPENSSSL\_DEBUG\_LEVEL” works.

**For example:** If OPENSSSL\_DEBUG\_LEVEL = 2, you use function “SSL\_DEBUG(1, “malloc failed”)”. Because  $1 < 2$ , it will not print.

## OPENSSSL\_LOWLEVEL\_DEBUG

Enable OpenSSL low-level module debugging

*Found in: Component config > OpenSSL*

If the option is enabled, low-level module debugging function of OpenSSL is enabled, e.g. mbedtls internal debugging function.

## OPENSSSL\_ASSERT

Select OpenSSL assert function

*Found in: Component config > OpenSSL*

OpenSSL function needs “assert” function to check if input parameters are valid.

If you want to use assert debugging function, “OPENSSSL\_DEBUG” should be enabled.

**Available options:**

- OPENSSSL\_ASSERT\_DO\_NOTHING
- OPENSSSL\_ASSERT\_EXIT
- OPENSSSL\_ASSERT\_DEBUG
- OPENSSSL\_ASSERT\_DEBUG\_EXIT
- OPENSSSL\_ASSERT\_DEBUG\_BLOCK

## AWS\_IOT\_SDK

Amazon Web Services IoT Platform

*Found in: Component config*

Select this option to enable support for the AWS IoT platform, via the esp-idf component for the AWS IoT Device C SDK.

## AWS\_IOT\_MQTT\_HOST

AWS IoT Endpoint Hostname

*Found in: Component config*

Default endpoint host name to connect to AWS IoT MQTT/S gateway

This is the custom endpoint hostname and is specific to an AWS IoT account. You can find it by logging into your AWS IoT Console and clicking the Settings button. The endpoint hostname is shown under the “Custom Endpoint” heading on this page.

If you need per-device hostnames for different regions or accounts, you can override the default hostname in your app.

## AWS\_IOT\_MQTT\_PORT

AWS IoT MQTT Port

*Found in: Component config*

Default port number to connect to AWS IoT MQTT/S gateway

If you need per-device port numbers for different regions, you can override the default port number in your app.

## AWS\_IOT\_MQTT\_TX\_BUF\_LEN

MQTT TX Buffer Length

*Found in: Component config*

Maximum MQTT transmit buffer size. This is the maximum MQTT message length (including protocol overhead) which can be sent.

Sending longer messages will fail.

## AWS\_IOT\_MQTT\_RX\_BUF\_LEN

MQTT RX Buffer Length

*Found in: Component config*

Maximum MQTT receive buffer size. This is the maximum MQTT message length (including protocol overhead) which can be received.

Longer messages are dropped.

## AWS\_IOT\_MQTT\_NUM\_SUBSCRIBE\_HANDLERS

Maximum MQTT Topic Filters

*Found in: Component config*

Maximum number of concurrent MQTT topic filters.

## AWS\_IOT\_MQTT\_MIN\_RECONNECT\_WAIT\_INTERVAL

Auto reconnect initial interval (ms)

*Found in: Component config*

Initial delay before making first reconnect attempt, if the AWS IoT connection fails. Client will perform exponential backoff, starting from this value.

## AWS\_IOT\_MQTT\_MAX\_RECONNECT\_WAIT\_INTERVAL

Auto reconnect maximum interval (ms)

*Found in: Component config*

Maximum delay between reconnection attempts. If the exponentially increased delay interval reaches this value, the client will stop automatically attempting to reconnect.

## Bluetooth

### BT\_ENABLED

Bluetooth

*Found in: Component config > Bluetooth*

Select this option to enable Bluetooth and show the submenu with Bluetooth configuration choices.

### BTDM\_CONTROLLER\_PINNED\_TO\_CORE\_CHOICE

The cpu core which bluetooth controller run

*Found in: Component config > Bluetooth*

Specify the cpu core to run bluetooth controller. Can not specify no-affinity.

**Available options:**

- BTDM\_CONTROLLER\_PINNED\_TO\_CORE\_0
- BTDM\_CONTROLLER\_PINNED\_TO\_CORE\_1

### BTDM\_CONTROLLER\_HCI\_MODE\_CHOICE

HCI mode

*Found in: Component config > Bluetooth*

Specify HCI mode as VHCI or UART(H4)

**Available options:**

- BTDM\_CONTROLLER\_HCI\_MODE\_VHCI
- BTDM\_CONTROLLER\_HCI\_MODE\_UART\_H4

## HCI UART(H4) Options

### BT\_HCI\_UART\_NO

UART Number for HCI

*Found in: Component config > Bluetooth > HCI UART(H4) Options*

Uart number for HCI. The available uart is UART1 and UART2.

### BT\_HCI\_UART\_BAUDRATE

UART Baudrate for HCI

*Found in: Component config > Bluetooth > HCI UART(H4) Options*

UART Baudrate for HCI. Please use standard baudrate.

### BLUEDROID\_ENABLED

Bluedroid Enable

*Found in: Component config > Bluetooth*

This enables the default Bluedroid Bluetooth stack

### BLUEDROID\_PINNED\_TO\_CORE\_CHOICE

The cpu core which Bluedroid run

*Found in: Component config > Bluetooth*

Which the cpu core to run Bluedroid. Can choose core0 and core1. Can not specify no-affinity.

**Available options:**

- BLUEDROID\_PINNED\_TO\_CORE\_0
- BLUEDROID\_PINNED\_TO\_CORE\_1

### BTC\_TASK\_STACK\_SIZE

Bluetooth event (callback to application) task stack size

*Found in: Component config > Bluetooth*

This select btc task stack size

### BLUEDROID\_MEM\_DEBUG

Bluedroid memory debug

*Found in: Component config > Bluetooth*

Bluedroid memory debug

## **CLASSIC\_BT\_ENABLED**

Classic Bluetooth

*Found in: Component config > Bluetooth*

For now this option needs “SMP\_ENABLE” to be set to yes

## **A2DP\_SINK\_TASK\_STACK\_SIZE**

A2DP sink (audio stream decoding) task stack size

*Found in: Component config > Bluetooth*

This affects the stack size of A2DP sink task which invokes the data callback function.

## **GATTS\_ENABLE**

Include GATT server module(GATTS)

*Found in: Component config > Bluetooth*

This option can be disabled when the app work only on gatt client mode

## **GATTC\_ENABLE**

Include GATT client module(GATTC)

*Found in: Component config > Bluetooth*

This option can be close when the app work only on gatt server mode

## **BLE\_SMP\_ENABLE**

Include BLE security module(SMP)

*Found in: Component config > Bluetooth*

This option can be close when the app not used the ble security connect.

## **BT\_STACK\_NO\_LOG**

Close the bluedroid bt stack log print

*Found in: Component config > Bluetooth*

This select can save the rodata code size

## **BT\_ACL\_CONNECTIONS**

BT/BLE MAX ACL CONNECTIONS(1~7)

*Found in: Component config > Bluetooth*

Maximum BT/BLE connection count

## BLE\_SCAN\_DUPLICATE

BLE Scan Duplicate Options

*Found in: Component config > Bluetooth*

This select enables parameters setting of BLE scan duplicate.

## DUPLICATE\_SCAN\_CACHE\_SIZE

Maximum number of devices in scan duplicate filter

*Found in: Component config > Bluetooth*

Maximum number of devices which can be recorded in scan duplicate filter. When the maximum amount of device in the filter is reached, the cache will be refreshed.

## BLE\_MESH\_SCAN\_DUPLICATE\_EN

Special duplicate scan mechanism for BLE Mesh scan

*Found in: Component config > Bluetooth*

This enables the BLE scan duplicate for special BLE Mesh scan.

## MESH\_DUPLICATE\_SCAN\_CACHE\_SIZE

Maximum number of Mesh adv packets in scan duplicate filter

*Found in: Component config > Bluetooth*

Maximum number of adv packets which can be recorded in duplicate scan cache for BLE Mesh. When the maximum amount of device in the filter is reached, the cache will be refreshed.

## BLE\_ACTIVE\_SCAN\_REPORT\_ADV\_SCAN\_RSP\_INDIVIDUALLY

Report adv data and scan response individually when BLE active scan

*Found in: Component config > Bluetooth*

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.

## mbedTLS

### 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).

## 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.

## 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.

## MBEDTLS\_HARDWARE\_MPI

Enable hardware MPI (bignum) acceleration

*Found in: Component config > mbedTLS*

Enable hardware accelerated multiple precision integer operations.

Hardware accelerated multiplication, modulo multiplication, and modular exponentiation for up to 4096 bit results.

These operations are used by RSA.

## MBEDTLS\_MPI\_USE\_INTERRUPT

Use interrupt for MPI operations

*Found in: Component config > mbedTLS*

Use an interrupt to coordinate MPI operations.

This allows other code to run on the CPU while an MPI operation is pending. Otherwise the CPU busy-waits.

## MBEDTLS\_HARDWARE\_SHA

Enable hardware SHA acceleration

*Found in: Component config > mbedTLS*

Enable hardware accelerated SHA1, SHA256, SHA384 & SHA512 in mbedTLS.

Due to a hardware limitation, hardware acceleration is only guaranteed if SHA digests are calculated one at a time. If more than one SHA digest is calculated at the same time, one will be calculated fully in hardware and the rest will be calculated (at least partially calculated) in software. This happens automatically.

SHA hardware acceleration is faster than software in some situations but slower in others. You should benchmark to find the best setting for you.

## MBEDTLS\_HAVE\_TIME

Enable mbedtls time

*Found in: Component config > mbedTLS*

System has time.h and time(). The time does not need to be correct, only time differences are used,

## MBEDTLS\_HAVE\_TIME\_DATE

Enable mbedtls time data

*Found in: Component config > mbedTLS*

System has time.h and time(), gmtime() and the clock is correct. The time needs to be correct (not necessarily very accurate, but at least the date should be correct). This is used to verify the validity period of X.509 certificates.

It is suggested that you should get the real time by “SNTP”.

## 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:**

- MBEDTLS\_TLS\_SERVER\_AND\_CLIENT
- MBEDTLS\_TLS\_SERVER\_ONLY
- MBEDTLS\_TLS\_CLIENT\_ONLY
- MBEDTLS\_TLS\_DISABLED

## TLS Key Exchange Methods

### 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.

### MBEDTLS\_KEY\_EXCHANGE\_PSK

Enable PSK based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support symmetric key PSK (pre-shared-key) TLS key exchange modes.

### MBEDTLS\_KEY\_EXCHANGE\_DHE\_PSK

Enable DHE-PSK based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

### MBEDTLS\_KEY\_EXCHANGE\_ECDHE\_PSK

Enable ECDHE-PSK based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support Elliptic-Curve-Diffie-Hellman PSK (pre-shared-key) TLS authentication modes.

### MBEDTLS\_KEY\_EXCHANGE\_RSA\_PSK

Enable RSA-PSK based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support RSA PSK (pre-shared-key) TLS authentication modes.

### 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-

### **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-

### **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.

### **MBEDTLS\_KEY\_EXCHANGE\_ECDHE\_RSA**

Enable ECDHE-RSA based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

### **MBEDTLS\_KEY\_EXCHANGE\_ECDHE\_ECDSA**

Enable ECDHE-ECDSA based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

### **MBEDTLS\_KEY\_EXCHANGE\_ECDH\_ECDSA**

Enable ECDH-ECDSA based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

### **MBEDTLS\_KEY\_EXCHANGE\_ECDH\_RSA**

Enable ECDH-RSA based ciphersuite modes

*Found in: Component config > mbedTLS > TLS Key Exchange Methods*

Enable to support ciphersuites with prefix TLS-ECDHE-RSA-WITH-

## **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.

## **MBEDTLS\_SSL\_PROTO\_SSL3**

Legacy SSL 3.0 support

*Found in: Component config > mbedTLS*

Support the legacy SSL 3.0 protocol. Most servers will speak a newer TLS protocol these days.

## **MBEDTLS\_SSL\_PROTO\_TLS1**

Support TLS 1.0 protocol

*Found in: Component config > mbedTLS*

## **MBEDTLS\_SSL\_PROTO\_TLS1\_1**

Support TLS 1.1 protocol

*Found in: Component config > mbedTLS*

## **MBEDTLS\_SSL\_PROTO\_TLS1\_2**

Support TLS 1.2 protocol

*Found in: Component config > mbedTLS*

## **MBEDTLS\_SSL\_PROTO\_DTLS**

Support DTLS protocol (all versions)

*Found in: Component config > mbedTLS*

Requires TLS 1.1 to be enabled for DTLS 1.0 Requires TLS 1.2 to be enabled for DTLS 1.2

## **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.

## MBEDTLS\_SSL\_SESSION\_TICKETS

TLS: Support RFC 5077 SSL session tickets

*Found in: Component config > mbedTLS*

Support RFC 5077 session tickets. See mbedTLS documentation for more details.

Disabling this option will save some code size.

## Symmetric Ciphers

### MBEDTLS\_AES\_C

AES block cipher

*Found in: Component config > mbedTLS > Symmetric Ciphers*

### MBEDTLS\_CAMELLIA\_C

Camellia block cipher

*Found in: Component config > mbedTLS > Symmetric Ciphers*

### 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.

### MBEDTLS\_RC4\_MODE

RC4 Stream Cipher (legacy, insecure)

*Found in: Component config > mbedTLS > Symmetric Ciphers*

ARCFOUR (RC4) stream cipher can be disabled entirely, enabled but not added to default ciphersuites, or enabled completely.

Please consider the security implications before enabling RC4.

**Available options:**

- MBEDTLS\_RC4\_DISABLED
- MBEDTLS\_RC4\_ENABLED\_NO\_DEFAULT
- MBEDTLS\_RC4\_ENABLED

## **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.

## **MBEDTLS\_XTEA\_C**

XTEA block cipher

*Found in: Component config > mbedTLS > Symmetric Ciphers*

Enables the XTEA block cipher.

## **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.

## **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.

## **MBEDTLS\_RIPEMD160\_C**

Enable RIPEMD-160 hash algorithm

*Found in: Component config > mbedTLS*

Enable the RIPEMD-160 hash algorithm.

## **Certificates**

### **MBEDTLS\_PEM\_PARSE\_C**

Read & Parse PEM formatted certificates

*Found in: Component config > mbedTLS > Certificates*

Enable decoding/parsing of PEM formatted certificates.

If your certificates are all in the simpler DER format, disabling this option will save some code size.

### **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.

### **MBEDTLS\_X509\_CRL\_PARSE\_C**

X.509 CRL parsing

*Found in: Component config > mbedTLS > Certificates*

Support for parsing X.509 Certificate Revocation Lists.

### **MBEDTLS\_X509\_CSR\_PARSE\_C**

X.509 CSR parsing

*Found in: Component config > mbedTLS > Certificates*

Support for parsing X.509 Certificate Signing Requests

### **MBEDTLS\_ECP\_C**

Elliptic Curve Ciphers

*Found in: Component config > mbedTLS*

### **MBEDTLS\_ECDH\_C**

Elliptic Curve Diffie-Hellman (ECDH)

*Found in: Component config > mbedTLS*

Enable ECDH. Needed to use ECDHE-xxx TLS ciphersuites.

### **MBEDTLS\_ECDSA\_C**

Elliptic Curve DSA

*Found in: Component config > mbedTLS*

Enable ECDSA. Needed to use ECDSA-xxx TLS ciphersuites.

### **MBEDTLS\_ECP\_DP\_SECP192R1\_ENABLED**

Enable SECP192R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP192R1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP224R1\_ENABLED**

Enable SECP224R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP224R1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP256R1\_ENABLED**

Enable SECP256R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP256R1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP384R1\_ENABLED**

Enable SECP384R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP384R1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP521R1\_ENABLED**

Enable SECP521R1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP521R1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP192K1\_ENABLED**

Enable SECP192K1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP192K1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP224K1\_ENABLED**

Enable SECP224K1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP224K1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_SECP256K1\_ENABLED**

Enable SECP256K1 curve

*Found in: Component config > mbedTLS*

Enable support for SECP256K1 Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_BP256R1\_ENABLED**

Enable BP256R1 curve

*Found in: Component config > mbedTLS*

support for DP Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_BP384R1\_ENABLED**

Enable BP384R1 curve

*Found in: Component config > mbedTLS*

support for DP Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_BP512R1\_ENABLED**

Enable BP512R1 curve

*Found in: Component config > mbedTLS*

support for DP Elliptic Curve.

### **MBEDTLS\_ECP\_DP\_CURVE25519\_ENABLED**

Enable CURVE25519 curve

*Found in: Component config > mbedTLS*

Enable support for CURVE25519 Elliptic Curve.

### **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.

## FAT Filesystem support

### 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:

- FATFS\_CODEPAGE\_DYNAMIC
- FATFS\_CODEPAGE\_437
- FATFS\_CODEPAGE\_720
- FATFS\_CODEPAGE\_737
- FATFS\_CODEPAGE\_771
- FATFS\_CODEPAGE\_775
- FATFS\_CODEPAGE\_850
- FATFS\_CODEPAGE\_852
- FATFS\_CODEPAGE\_855
- FATFS\_CODEPAGE\_857
- FATFS\_CODEPAGE\_860
- FATFS\_CODEPAGE\_861
- FATFS\_CODEPAGE\_862
- FATFS\_CODEPAGE\_863
- FATFS\_CODEPAGE\_864
- FATFS\_CODEPAGE\_865
- FATFS\_CODEPAGE\_866
- FATFS\_CODEPAGE\_869
- FATFS\_CODEPAGE\_932
- FATFS\_CODEPAGE\_936
- FATFS\_CODEPAGE\_949
- FATFS\_CODEPAGE\_950

### 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:**

- FATFS\_LFN\_NONE
- FATFS\_LFN\_HEAP
- FATFS\_LFN\_STACK

## FATFS\_MAX\_LFN

Max long filename length

*Found in: Component config > FAT Filesystem support*

Maximum long filename length. Can be reduced to save RAM.

## 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.

## FATFS\_TIMEOUT\_MS

Timeout for acquiring a file lock, ms

*Found in: Component config > FAT Filesystem support*

This option sets FATFS configuration value `_FS_TIMEOUT`, scaled to milliseconds. Sets the number of milliseconds FATFS will wait to acquire a mutex when operating on an open file. For example, if one task is performing a lengthy operation, another task will wait for the first task to release the lock, and time out after amount of time set by this option.

## 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.

## SPIFFS Configuration

### SPIFFS\_MAX\_PARTITIONS

Maximum Number of Partitions

*Found in: Component config > SPIFFS Configuration*

Define maximum number of partitions that can be mounted.

## SPIFFS Cache Configuration

### SPIFFS\_CACHE

Enable SPIFFS Cache

*Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration*

Enables/disable memory read caching of nucleus file system operations.

### SPIFFS\_CACHE\_WR

Enable SPIFFS Write Caching

*Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration*

Enables memory write caching for file descriptors in hydrogen.

### SPIFFS\_CACHE\_STATS

Enable SPIFFS Cache Statistics

*Found in: Component config > SPIFFS Configuration > SPIFFS Cache Configuration*

Enable/disable statistics on caching. Debug/test purpose only.

### 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, will increase flash.

## 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.

## SPIFFS\_GC\_STATS

Enable SPIFFS GC Statistics

*Found in: Component config > SPIFFS Configuration*

Enable/disable statistics on gc. Debug/test purpose only.

## 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\_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 on mount point.

## SPIFFS\_USE\_MAGIC\_LENGTH

Enable SPIFFS Filesystem Length Magic

*Found in: Component config > SPIFFS Configuration*

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.

## 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\_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.

## **Debug Configuration**

### **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

### **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

### **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

### **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

### **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

## 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.

## SPI Flash driver

### 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.

### SPI\_FLASH\_LOG\_FAILED\_WRITE

Log errors if verification fails

*Found in: Component config > SPI Flash driver*

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.

### SPI\_FLASH\_WARN\_SETTING\_ZERO\_TO\_ONE

Log warning if writing zero bits to ones

*Found in: Component config > SPI Flash driver*

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.

### SPI\_FLASH\_ENABLE\_COUNTERS

Enable operation counters

*Found in: Component config > SPI Flash driver*

This option enables the following APIs:

- spi\_flash\_reset\_counters
- spi\_flash\_dump\_counters
- spi\_flash\_get\_counters

These APIs may be used to collect performance data for spi\_flash APIs and to help understand behaviour of libraries which use SPI flash.

## SPI\_FLASH\_ROM\_DRIVER\_PATCH

Enable SPI flash ROM driver patched functions

*Found in: Component config > SPI Flash driver*

Enable this flag to use patched versions of SPI flash ROM driver functions. This option is needed to write to flash on ESP32-D2WD, and any configuration where external SPI flash is connected to non-default pins.

## SPI\_FLASH\_WRITING\_DANGEROUS\_REGIONS

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:

- SPI\_FLASH\_WRITING\_DANGEROUS\_REGIONS\_ABORTS
- SPI\_FLASH\_WRITING\_DANGEROUS\_REGIONS\_FAILS
- SPI\_FLASH\_WRITING\_DANGEROUS\_REGIONS\_ALLOWED

## Serial flasher config

### ESPTOOLPY\_PORT

Default serial port

*Found in: Serial flasher config*

The serial port that's connected to the ESP chip. This can be overridden by setting the ESPPORT environment variable.

### ESPTOOLPY\_BAUD

Default baud rate

*Found in: Serial flasher config*

Default baud rate to use while communicating with the ESP chip. Can be overridden by setting the ESPBAUD variable.

**Available options:**

- ESPTOOLPY\_BAUD\_115200B
- ESPTOOLPY\_BAUD\_230400B
- ESPTOOLPY\_BAUD\_921600B
- ESPTOOLPY\_BAUD\_2MB
- ESPTOOLPY\_BAUD\_OTHER

## ESPTOOLPY\_BAUD\_OTHER\_VAL

Other baud rate value

*Found in: Serial flasher config*

## ESPTOOLPY\_COMPRESSED

Use compressed upload

*Found in: Serial flasher config*

The flasher tool can send data compressed using zlib, letting the ROM on the ESP chip decompress it on the fly before flashing it. For most payloads, this should result in a speed increase.

## 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:**

- FLASHMODE\_QIO
- FLASHMODE\_QOUT
- FLASHMODE\_DIO
- FLASHMODE\_DOUT

## ESPTOOLPY\_FLASHFREQ

Flash SPI speed

*Found in: Serial flasher config*

The SPI flash frequency to be used.

**Available options:**

- ESPTOOLPY\_FLASHFREQ\_80M
- ESPTOOLPY\_FLASHFREQ\_40M
- ESPTOOLPY\_FLASHFREQ\_26M

- ESPTOOLPY\_FLASHFREQ\_20M

## ESPTOOLPY\_FLASHSIZE

Flash size

*Found in: Serial flasher config*

SPI flash size, in megabytes

### Available options:

- ESPTOOLPY\_FLASHSIZE\_1MB
- ESPTOOLPY\_FLASHSIZE\_2MB
- ESPTOOLPY\_FLASHSIZE\_4MB
- ESPTOOLPY\_FLASHSIZE\_8MB
- ESPTOOLPY\_FLASHSIZE\_16MB

## ESPTOOLPY\_FLASHSIZE\_DETECT

Detect flash size when flashing bootloader

*Found in: Serial flasher config*

If this option is set, ‘make flash’ targets will automatically detect the flash size and update the bootloader image when flashing.

## 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:

- ESPTOOLPY\_BEFORE\_RESET
- ESPTOOLPY\_BEFORE\_NORESET

## 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:

- ESPTOOLPY\_AFTER\_RESET
- ESPTOOLPY\_AFTER\_NORESET

## MONITOR\_BAUD

‘make monitor’ baud rate

*Found in: Serial flasher config*

Baud rate to use when running ‘make monitor’ to view serial output from a running chip.

Can override by setting the MONITORBAUD environment variable.

### Available options:

- MONITOR\_BAUD\_9600B
- MONITOR\_BAUD\_57600B
- MONITOR\_BAUD\_115200B
- MONITOR\_BAUD\_230400B
- MONITOR\_BAUD\_921600B
- MONITOR\_BAUD\_2MB
- MONITOR\_BAUD\_OTHER

## MONITOR\_BAUD\_OTHER\_VAL

Custom baud rate value

*Found in: Serial flasher config*

## tcpip adapter

### IP\_LOST\_TIMER\_INTERVAL

IP Address lost timer interval (seconds)

*Found in: Component config > tcpip 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.

## 2.8.4 Customisations

Because IDF builds by default with *Warning On Undefined Variables*, when the Kconfig tool generates Makefiles (the `auto.conf` file) its behaviour has been customised. In normal Kconfig, a variable which is set to “no” is undefined. In IDF’s version of Kconfig, this variable is defined in the Makefile but has an empty value.

(Note that `ifdef` and `ifndef` can still be used in Makefiles, because they test if a variable is defined *and has a non-empty value*.)

When generating header files for C & C++, the behaviour is not customised - so `#ifdef` can be used to test if a boolean config item is set or not.



## 3.1 ESP32 Modules and Boards

Espressif designed and manufactured several development modules and boards to help users evaluate functionality of the ESP32 family of chips. Development boards, depending on intended functionality, have exposed GPIO pins headers, provide USB programming interface, JTAG interface as well as peripherals like touch pads, LCD screen, SD card slot, camera module header, etc.

For details please refer to documentation below, provided together with description of particular boards.

**Note:** This section describes the latest versions of boards. Previous versions of boards, including these not produced anymore, are described in section *Previous Versions of ESP32 Modules and Boards*.

### 3.1.1 WROOM and WROVER Modules

A family of small modules that contain ESP32 chip on board together with some key components including a crystal oscillator and an antenna matching circuit. This makes it easier to provide an ESP32 based solution ready to integrate into final products. Such modules can be also used for evaluation after adding a few extra components like a programming interface, bootstrapping resistors and break out headers. The key characteristics of these modules are summarized in the following table. Some additional details are covered in the following chapters.

Module	Key Components				Dimensions [mm]		
	Chip	Flash	RAM	Ant.	L	W	D
ESP-WROOM-32	ESP32-D0WDQ6	4MB	–	MIFA	25.5	18	3.1
ESP-WROOM-32D	ESP32-D0WD	4MB	–	MIFA	25.5	18	3.1
ESP32-WROOM-32U	ESP32-D0WD	4MB	–	U.FL	19.2	18	3.2
ESP32-WROVER	ESP32-D0WDQ6	4MB	4MB	MIFA	31.4	18	3.2
ESP32-WROVER-I	ESP32-D0WDQ6	4MB	4MB	U.FL	31.4	18	3.5

- MIFA - Meandered Inverted-F Antenna





Fig. 3: ESP32-WROOM-32U module (back and front)

## Documentation

- [ESP-WROOM-32D / ESP32-WROOM-32U Datasheet \(PDF\)](#)

## ESP32-WROVER

A step upgrade of *ESP-WROOM-32* with an additional 4 MB SPI PSRAM (Pseudo static RAM). This module is provided in two versions: ‘ESP32-WROVER’ with PCB antenna (shown below) and ‘ESP32-WROVER-I’ with an U.FL / IPEX antenna connector. Because of additional components inside, this module is 5.9 mm longer than *ESP-WROOM-32*.



Fig. 4: ESP32-WROVER module (front and back)

## Documentation

- [ESP32-WROVER Datasheet \(PDF\)](#)
- [ESP-PSRAM32 Datasheet \(PDF\)](#)

### 3.1.2 ESP32-PICO-KIT V4

The smallest ESP32 development board with all the components required to connect it directly to a PC USB port, and pin headers to plug into a mini breadboard. It is equipped with ESP32-PICO-D4 chip that integrates 4MB flash memory, a crystal oscillator, filter capacitors and RF matching circuit in one single package. As result the fully functional development board requires only a few external components that can easy fit on a 20 x 52 mm PCB including antenna, LDO, USB-UART bridge and two buttons to reset it and put into download mode.

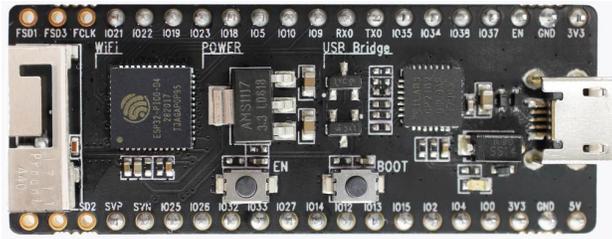


Fig. 5: ESP32-PICO-KIT V4 board

Comparing to ESP32-PICO-KIT V3, this version has revised printout and reduced number of exposed pins. Instead of 20, only 17 header pins are populated, so V4 can fit into a mini breadboard.

## Documentation

- [ESP32-PICO-KIT V4 Getting Started Guide](#)
- [ESP32-PICO-KIT V4 Schematic \(PDF\)](#)
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)

## Previous Versions

- [ESP32-PICO-KIT V3](#)

### 3.1.3 ESP32 Core Board V2 / ESP32 DevKitC

Small and convenient development board with *ESP-WROOM-32* module installed, break out pin headers and minimum additional components. Includes USB to serial programming interface, that also provides power supply for the board. Has pushbuttons to reset the board and put it in upload mode.



Fig. 6: ESP32 Core Board V2 / ESP32 DevKitC board

## Documentation

- [ESP32-DevKitC Getting Started Guide](#)
- [ESP32 DevKitC Schematic \(PDF\)](#)
- [ESP32 Development Board Reference Design \(ZIP\)](#) containing OrCAD schematic, PCB layout, gerbers and BOM

- [CP210x USB to UART Bridge VCP Drivers](#)

### 3.1.4 ESP-WROVER-KIT V3

The ESP-WROVER-KIT V3 development board has dual port USB to serial converter for programming and JTAG interface for debugging. Power supply is provided by USB interface or from standard 5 mm power supply jack. Power supply selection is done with a jumper and may be put on/off with a separate switch. This board has MicroSD card slot, 3.2" SPI LCD screen and dedicated header to connect a camera. It provides RGB diode for diagnostics. Includes 32.768 kHz XTAL for internal RTC to operate it in low power modes.

As all previous version of ESP-WROVER-KIT boards, it is ready to accommodate an *ESP-WROOM-32* or *ESP32-WROVER* module.

This is the first release of ESP-WROVER-KIT shipped with *ESP32-WROVER* module installed by default. This release also introduced several design changes to conditioning and interlocking of signals to the bootstrapping pins. Also, a zero Ohm resistor (R166) has been added between WROVER/WROOM module and VDD33 net, which can be desoldered, or replaced with a shunt resistor, for current measurement. This is intended to facilitate power consumption analysis in various operation modes of ESP32. Refer to schematic - the changes are enclosed in green border.

The camera header has been changed from male back to female. The board soldermask is matte black. The board on picture above has *ESP32-WROVER* is installed.

#### Documentation

- [ESP-WROVER-KIT V3 Getting Started Guide](#)
- [ESP-WROVER-KIT V3 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

#### Previous Versions

- [ESP-WROVER-KIT V1 / ESP32 DevKitJ V1](#)
- [ESP-WROVER-KIT V2](#)

### 3.1.5 Related Documents

- [Previous Versions of ESP32 Modules and Boards](#)

## 3.2 Previous Versions of ESP32 Modules and Boards

This sections contains overview and links to documentation of previous version ESP32 Modules and Boards that have been replaced with newer versions or discontinued. It is maintained for convenience of users as several of these boards are still in use and some may still be available for purchase.

To see the latest development boards, please refer to section [ESP32 Modules and Boards](#).

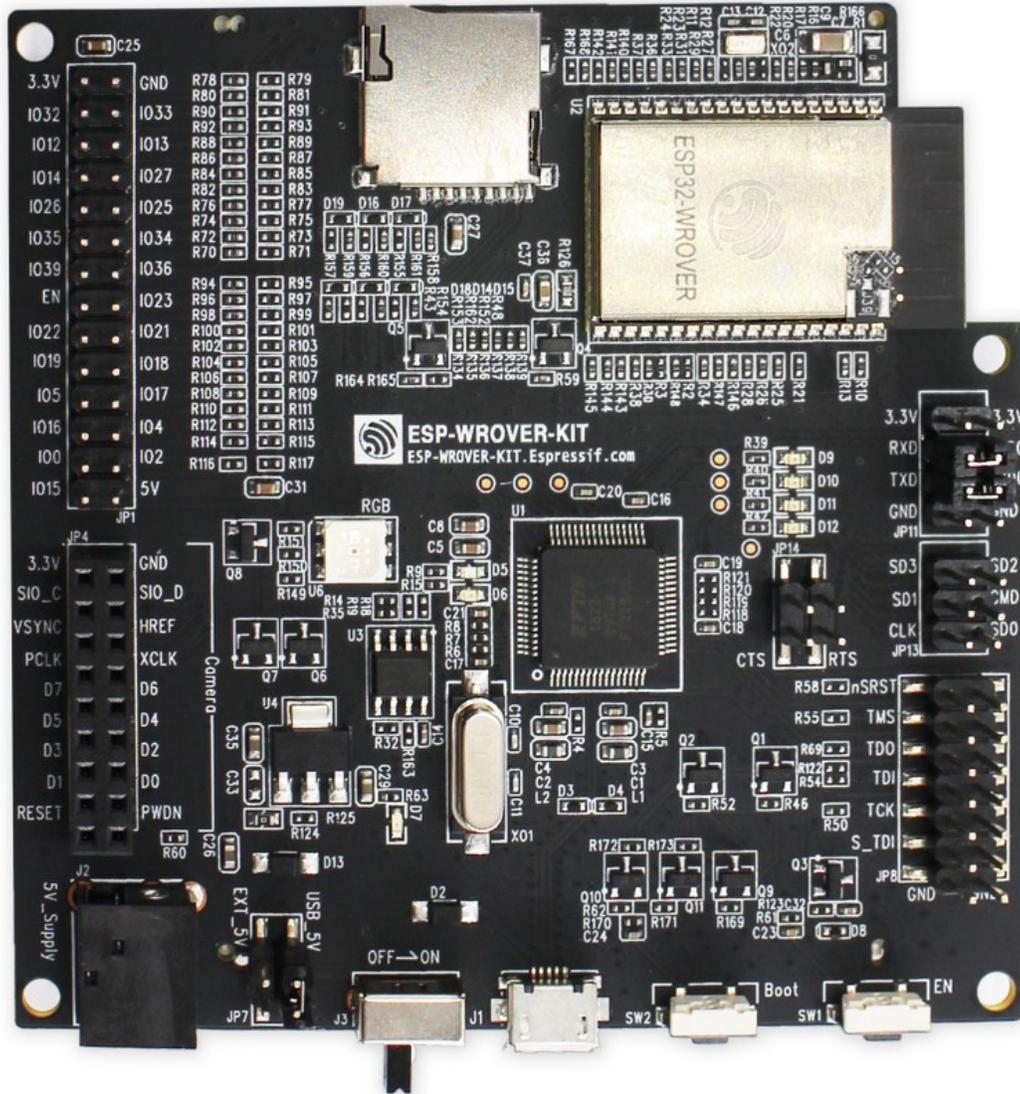


Fig. 7: ESP-WROVER-KIT V3 board

### 3.2.1 ESP32-PICO-KIT V3

The first public release of Espressif's ESP32-PICO-D4 chip on a mini development board. The board has a USB port for programming and debugging and two rows of 20 pin headers to plug into a breadboard. The ESP32-PICO-D4 chip itself is small and requires only a few external components. Besides two core CPUs it integrates 4MB flash memory, a crystal oscillator and antenna matching components in one single 7 x 7 mm package. As a result the chip and all the components making the complete development board fit into 20 x 52 mm PCB.



Fig. 8: ESP32-PICO-KIT V3 board

#### Documentation

- [ESP32-PICO-KIT V3 Getting Started Guide](#)
- [ESP32-PICO-KIT V3 Schematic \(PDF\)](#)
- [ESP32-PICO-D4 Datasheet \(PDF\)](#)

### 3.2.2 ESP-WROVER-KIT V1 / ESP32 DevKitJ V1

The first version of ESP-WROVER-KIT development board. Shipped with ESP-WROOM-32 on board.

ESP-WROVER-KIT has dual port USB to serial converter for programming and JTAG interface for debugging. Power supply is provided by USB interface or from standard 5 mm power supply jack. Power supply selection is done with a jumper and may be put on/off with a separate switch. The board has MicroSD card slot, 3.2" SPI LCD screen and dedicated header to connect a camera. It provides RGB diode for diagnostics. Includes 32.768 kHz XTAL for internal RTC to operate it in low power modes.

All versions of ESP-WROVER-KIT are ready to accommodate an [ESP-WROOM-32](#) or [ESP32-WROVER](#) module.

The board has red soldermask.

#### Documentation

- [ESP-WROVER-KIT V1 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

### 3.2.3 ESP-WROVER-KIT V2

This is updated version of ESP32 DevKitJ V1 described above with design improvements identified when DevKitJ was in use, e.g. improved support for SD card. By default board has ESP-WROOM-32 module installed.

Comparing to previous version, this board has a shiny black finish and a male camera header.

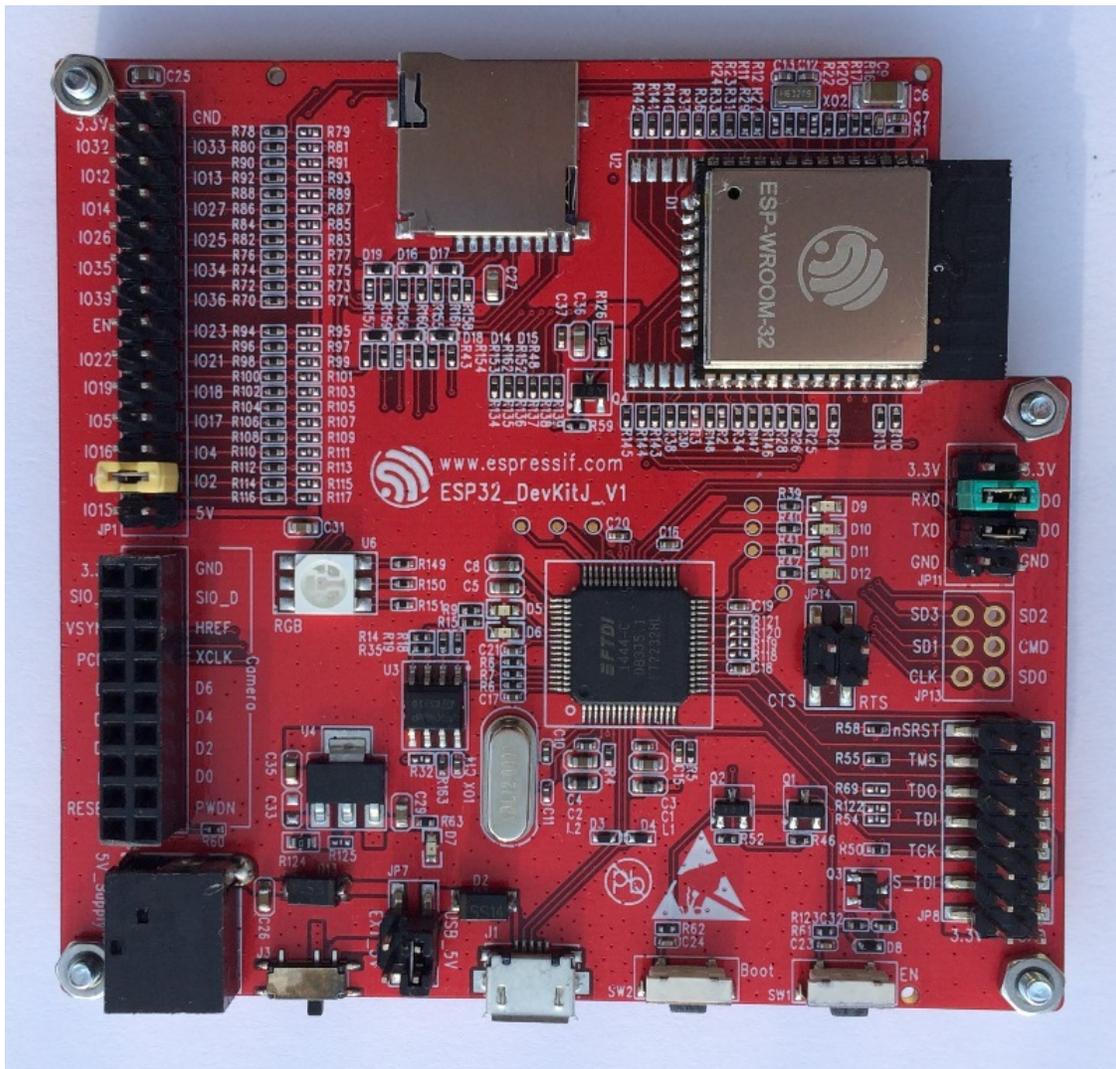


Fig. 9: ESP-WROVER-KIT V1 / ESP32 DevKitJ V1 board

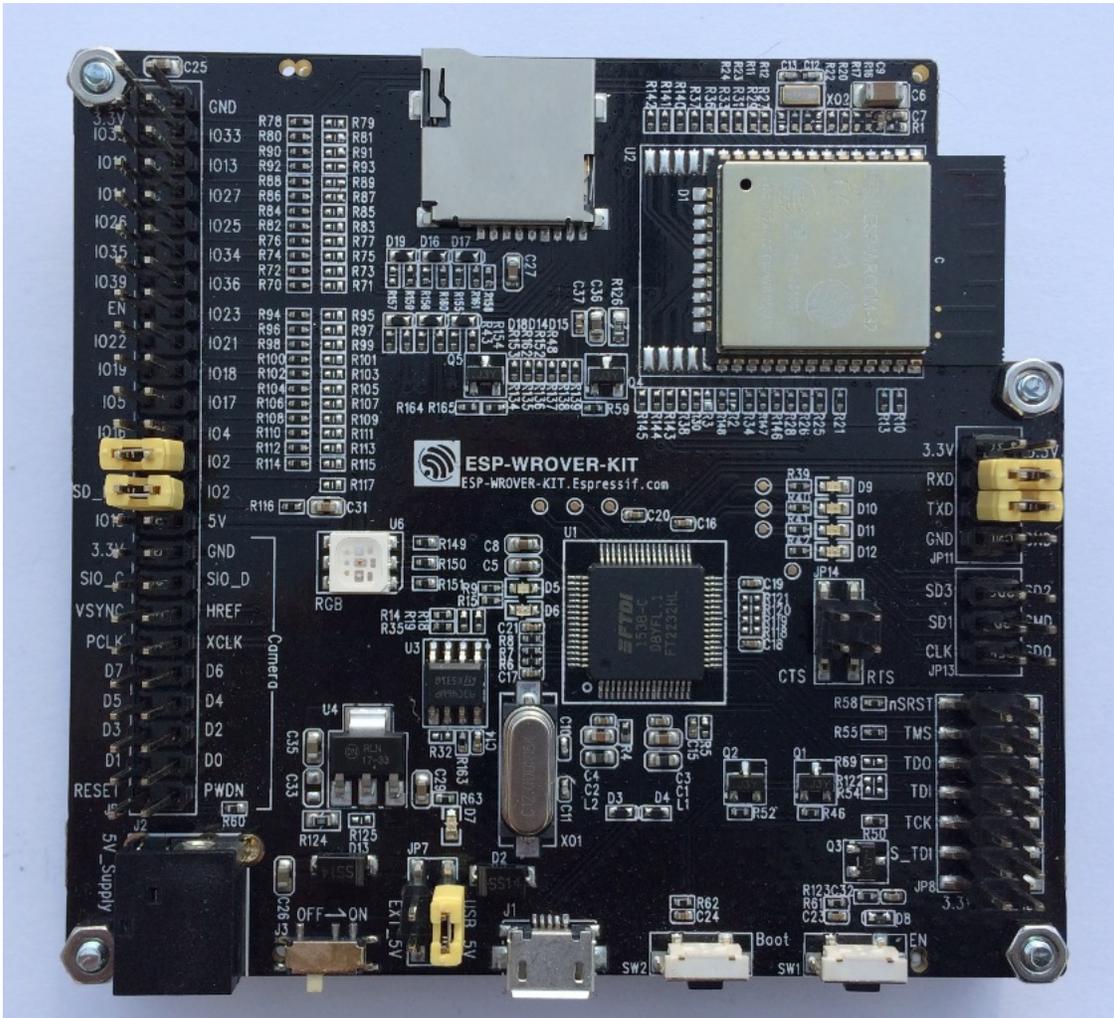


Fig. 10: ESP-WROVER-KIT V2 board

## Documentation

- [ESP-WROVER-KIT V2 Getting Started Guide](#)
- [ESP-WROVER-KIT V2 Schematic \(PDF\)](#)
- [JTAG Debugging](#)
- [FTDI Virtual COM Port Drivers](#)

### 3.2.4 ESP32 Demo Board V2

One of first feature rich evaluation boards that contains several pin headers, dip switches, USB to serial programming interface, reset and boot mode press buttons, power switch, 10 touch pads and separate header to connect LCD screen.

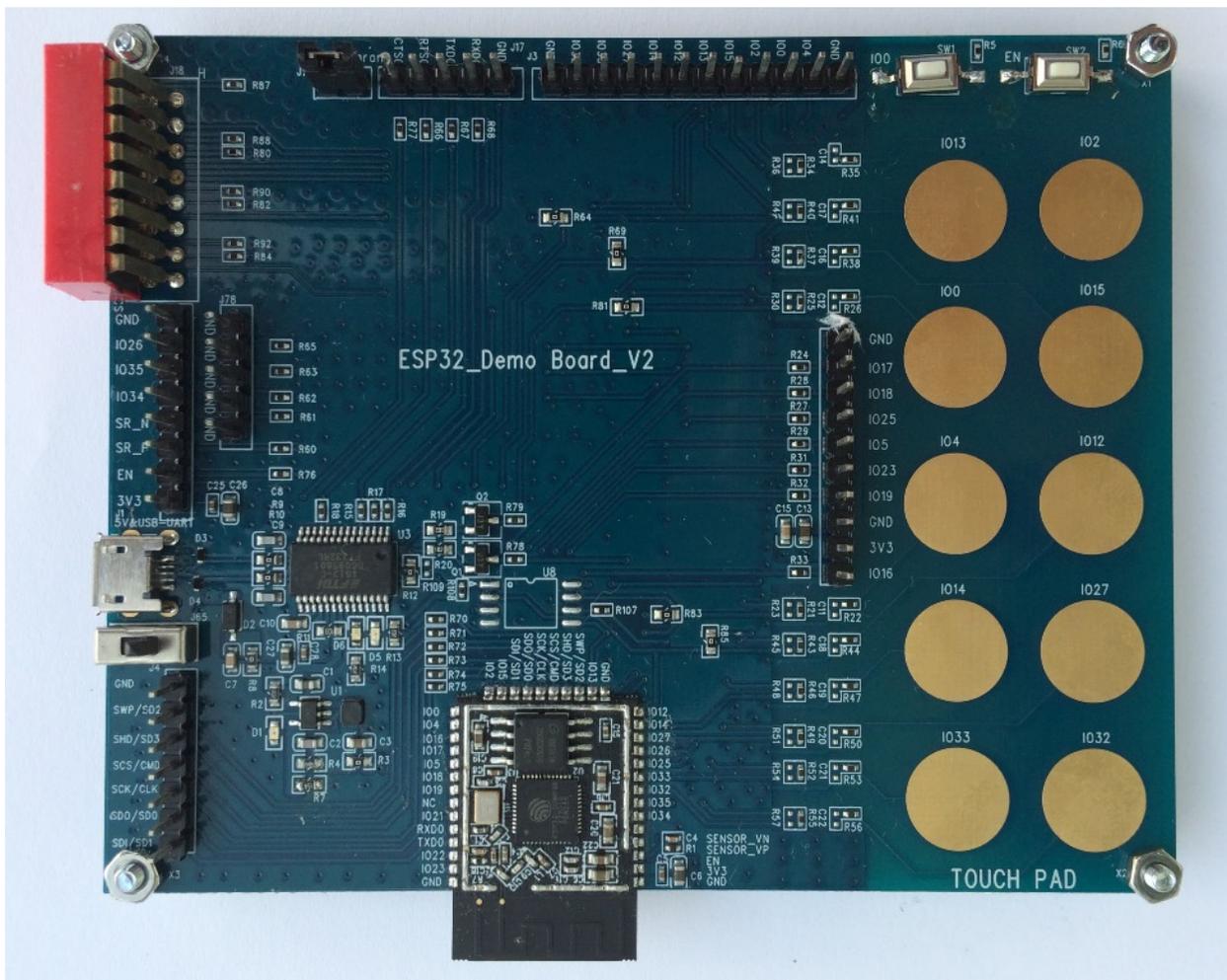


Fig. 11: ESP32 Demo Board V2

Production of this board is discontinued.

## Documentation

- [ESP32 Demo Board V2 Schematic \(PDF\)](#)

- [FTDI Virtual COM Port Drivers](#)

### 3.2.5 Related Documents

- *[ESP32 Modules and Boards](#)*



## 4.1 General Notes About ESP-IDF Programming

### 4.1.1 Application startup flow

This note explains various steps which happen before `app_main` function of an ESP-IDF application is called.

The high level view of startup process is as follows:

1. First-stage bootloader in ROM loads second-stage bootloader image to RAM (IRAM & DRAM) from flash offset 0x1000.
2. Second-stage bootloader loads partition table and main app image from flash. Main app incorporates both RAM segments and read-only segments mapped via flash cache.
3. Main app image executes. At this point the second CPU and RTOS scheduler can be started.

This process is explained in detail in the following sections.

#### First stage bootloader

After SoC reset, PRO CPU will start running immediately, executing reset vector code, while APP CPU will be held in reset. During startup process, PRO CPU does all the initialization. APP CPU reset is de-asserted in the `call_start_cpu0` function of application startup code. Reset vector code is located at address 0x40000400 in the mask ROM of the ESP32 chip and can not be modified.

Startup code called from the reset vector determines the boot mode by checking `GPIO_STRAP_REG` register for bootstrap pin states. Depending on the reset reason, the following takes place:

1. Reset from deep sleep: if the value in `RTC_CNTL_STORE6_REG` is non-zero, and CRC value of RTC memory in `RTC_CNTL_STORE7_REG` is valid, use `RTC_CNTL_STORE6_REG` as an entry point address and jump immediately to it. If `RTC_CNTL_STORE6_REG` is zero, or `RTC_CNTL_STORE7_REG` contains invalid CRC, or once the code called via `RTC_CNTL_STORE6_REG` returns, proceed with boot as if it was a power-on reset. **Note:** to run customized code at this point, a deep sleep stub mechanism is provided. Please see [deep sleep](#) documentation for this.

2. For power-on reset, software SOC reset, and watchdog SOC reset: check the `GPIO_STRAP_REG` register if UART or SDIO download mode is requested. If this is the case, configure UART or SDIO, and wait for code to be downloaded. Otherwise, proceed with boot as if it was due to software CPU reset.
3. For software CPU reset and watchdog CPU reset: configure SPI flash based on EFUSE values, and attempt to load the code from flash. This step is described in more detail in the next paragraphs. If loading code from flash fails, unpack BASIC interpreter into the RAM and start it. Note that RTC watchdog is still enabled when this happens, so unless any input is received by the interpreter, watchdog will reset the SOC in a few hundred milliseconds, repeating the whole process. If the interpreter receives any input from the UART, it disables the watchdog.

Application binary image is loaded from flash starting at address 0x1000. First 4kB sector of flash is used to store secure boot IV and signature of the application image. Please check secure boot documentation for details about this.

## Second stage bootloader

In ESP-IDF, the binary image which resides at offset 0x1000 in flash is the second stage bootloader. Second stage bootloader source code is available in `components/bootloader` directory of ESP-IDF. Note that this arrangement is not the only one possible with the ESP32 chip. It is possible to write a fully featured application which would work when flashed to offset 0x1000, but this is out of scope of this document. Second stage bootloader is used in ESP-IDF to add flexibility to flash layout (using partition tables), and allow for various flows associated with flash encryption, secure boot, and over-the-air updates (OTA) to take place.

When the first stage bootloader is finished checking and loading the second stage bootloader, it jumps to the second stage bootloader entry point found in the binary image header.

Second stage bootloader reads the partition table found at offset 0x8000. See [partition tables](#) documentation for more information. The bootloader finds factory and OTA partitions, and decides which one to boot based on data found in *OTA info* partition.

For the selected partition, second stage bootloader copies data and code sections which are mapped into IRAM and DRAM to their load addresses. For sections which have load addresses in DROM and IROM regions, flash MMU is configured to provide the correct mapping. Note that the second stage bootloader configures flash MMU for both PRO and APP CPUs, but it only enables flash MMU for PRO CPU. Reason for this is that second stage bootloader code is loaded into the memory region used by APP CPU cache. The duty of enabling cache for APP CPU is passed on to the application. Once code is loaded and flash MMU is set up, second stage bootloader jumps to the application entry point found in the binary image header.

Currently it is not possible to add application-defined hooks to the bootloader to customize application partition selection logic. This may be required to load different application image depending on a state of a GPIO, for example. Such customization features will be added to ESP-IDF in the future. For now, bootloader can be customized by copying bootloader component into application directory and making necessary changes there. ESP-IDF build system will compile the component in application directory instead of ESP-IDF components directory in this case.

## Application startup

ESP-IDF application entry point is `call_start_cpu0` function found in `components/esp32/cpu_start.c`. Two main things this function does are to enable heap allocator and to make APP CPU jump to its entry point, `call_start_cpu1`. The code on PRO CPU sets the entry point for APP CPU, de-asserts APP CPU reset, and waits for a global flag to be set by the code running on APP CPU, indicating that it has started. Once this is done, PRO CPU jumps to `start_cpu0` function, and APP CPU jumps to `start_cpu1` function.

Both `start_cpu0` and `start_cpu1` are weak functions, meaning that they can be overridden in the application, if some application-specific change to initialization sequence is needed. Default implementation of `start_cpu0` enables or initializes components depending on choices made in `menuconfig`. Please see source code of this function in `components/esp32/cpu_start.c` for an up to date list of steps performed. Note that any C++ global

constructors present in the application will be called at this stage. Once all essential components are initialized, *main task* is created and FreeRTOS scheduler is started.

While PRO CPU does initialization in `start_cpu0` function, APP CPU spins in `start_cpu1` function, waiting for the scheduler to be started on the PRO CPU. Once the scheduler is started on the PRO CPU, code on the APP CPU starts the scheduler as well.

Main task is the task which runs `app_main` function. Main task stack size and priority can be configured in `menuconfig`. Application can use this task for initial application-specific setup, for example to launch other tasks. Application can also use main task for event loops and other general purpose activities. If `app_main` function returns, main task is deleted.

## 4.1.2 Application memory layout

ESP32 chip has flexible memory mapping features. This section describes how ESP-IDF uses these features by default. Application code in ESP-IDF can be placed into one of the following memory regions.

### IRAM (instruction RAM)

ESP-IDF allocates part of *Internal SRAM0* region (defined in the Technical Reference Manual) for instruction RAM. Except for the first 64 kB block which is used for PRO and APP CPU caches, the rest of this memory range (i.e. from `0x40080000` to `0x400A0000`) is used to store parts of application which need to run from RAM.

A few components of ESP-IDF and parts of WiFi stack are placed into this region using the linker script.

If some application code needs to be placed into IRAM, it can be done using `IRAM_ATTR` define:

```
#include "esp_attr.h"

void IRAM_ATTR gpio_isr_handler(void* arg)
{
    // ...
}
```

Here are the cases when parts of application may or should be placed into IRAM.

- Interrupt handlers must be placed into IRAM if `ESP_INTR_FLAG_IRAM` is used when registering the interrupt handler. In this case, ISR may only call functions placed into IRAM or functions present in ROM. *Note 1:* all FreeRTOS APIs are currently placed into IRAM, so are safe to call from interrupt handlers. If the ISR is placed into IRAM, all constant data used by the ISR and functions called from ISR (including, but not limited to, `const char` arrays), must be placed into DRAM using `DRAM_ATTR`.
- Some timing critical code may be placed into IRAM to reduce the penalty associated with loading the code from flash. ESP32 reads code and data from flash via a 32 kB cache. In some cases, placing a function into IRAM may reduce delays caused by a cache miss.

### IROM (code executed from Flash)

If a function is not explicitly placed into IRAM or RTC memory, it is placed into flash. The mechanism by which Flash MMU is used to allow code execution from flash is described in the Technical Reference Manual. ESP-IDF places the code which should be executed from flash starting from the beginning of `0x400D0000` -- `0x40400000` region. Upon startup, second stage bootloader initializes Flash MMU to map the location in flash where code is located into the beginning of this region. Access to this region is transparently cached using two 32kB blocks in `0x40070000` — `0x40080000` range.

Note that the code outside `0x40000000 -- 0x40400000` region may not be reachable with Window ABI `CALLx` instructions, so special care is required if `0x40400000 -- 0x40800000` or `0x40800000 -- 0x40C00000` regions are used by the application. ESP-IDF doesn't use these regions by default.

### RTC fast memory

The code which has to run after wake-up from deep sleep mode has to be placed into RTC memory. Please check detailed description in [deep sleep](#) documentation.

### DRAM (data RAM)

Non-constant static data and zero-initialized data is placed by the linker into the 256 kB `0x3FFB0000 -- 0x3FFF0000` region. Note that this region is reduced by 64kB (by shifting start address to `0x3FFC0000`) if Bluetooth stack is used. Length of this region is also reduced by 16 kB or 32kB if trace memory is used. All space which is left in this region after placing static data there is used for the runtime heap.

Constant data may also be placed into DRAM, for example if it is used in an ISR (see notes in IRAM section above). To do that, `DRAM_ATTR` define can be used:

```
DRAM_ATTR const char[] format_string = "%p %x";
char buffer[64];
sprintf(buffer, format_string, ptr, val);
```

Needless to say, it is not advised to use `printf` and other output functions in ISRs. For debugging purposes, use `ESP_EARLY_LOGx` macros when logging from ISRs. Make sure that both `TAG` and format string are placed into DRAM in that case.

### DROM (data stored in Flash)

By default, constant data is placed by the linker into a 4 MB region (`0x3F400000 -- 0x3F800000`) which is used to access external flash memory via Flash MMU and cache. Exceptions to this are literal constants which are embedded by the compiler into application code.

### RTC slow memory

Global and static variables used by code which runs from RTC memory (i.e. deep sleep stub code) must be placed into RTC slow memory. Please check detailed description in [deep sleep](#) documentation.

## 4.2 Build System

This document explains the Espressif IoT Development Framework build system and the concept of “components”

Read this document if you want to know how to organise a new ESP-IDF project.

We recommend using the [esp-idf-template](#) project as a starting point for your project.

### 4.2.1 Using the Build System

The `esp-idf` README file contains a description of how to use the build system to build your project.

## 4.2.2 Overview

An ESP-IDF project can be seen as an amalgamation of a number of components. For example, for a webserver that shows the current humidity, there could be:

- The ESP32 base libraries (libc, rom bindings etc)
- The WiFi drivers
- A TCP/IP stack
- The FreeRTOS operating system
- A webserver
- A driver for the humidity sensor
- Main code tying it all together

ESP-IDF makes these components explicit and configurable. To do that, when a project is compiled, the build environment will look up all the components in the ESP-IDF directories, the project directories and (optionally) in additional custom component directories. It then allows the user to configure the ESP-IDF project using a text-based menu system to customize each component. After the components in the project are configured, the build process will compile the project.

### Concepts

- A “project” is a directory that contains all the files and configuration to build a single “app” (executable), as well as additional supporting output such as a partition table, data/filesystem partitions, and a bootloader.
- “Project configuration” is held in a single file called `sdkconfig` in the root directory of the project. This configuration file is modified via `make menuconfig` to customise the configuration of the project. A single project contains exactly one project configuration.
- An “app” is an executable which is built by `esp-idf`. A single project will usually build two apps - a “project app” (the main executable, ie your custom firmware) and a “bootloader app” (the initial bootloader program which launches the project app).
- “components” are modular pieces of standalone code which are compiled into static libraries (`.a` files) and linked into an app. Some are provided by `esp-idf` itself, others may be sourced from other places.

Some things are not part of the project:

- “ESP-IDF” is not part of the project. Instead it is standalone, and linked to the project via the `IDF_PATH` environment variable which holds the path of the `esp-idf` directory. This allows the IDF framework to be decoupled from your project.
- The toolchain for compilation is not part of the project. The toolchain should be installed in the system command line `PATH`, or the path to the toolchain can be set as part of the compiler prefix in the project configuration.

### Example Project

An example project directory tree might look like this:

```

- myProject/
  - Makefile
  - sdkconfig
  - components/ - component1/ - component.mk
                    - Kconfig

```

(continues on next page)

(continued from previous page)

```

- src1.c
- component2/ - component.mk
- Kconfig
- src1.c
- include/ - component2.h
- main/ - src1.c
- src2.c
- component.mk
- build/

```

This example “myProject” contains the following elements:

- A top-level project Makefile. This Makefile set the `PROJECT_NAME` variable and (optionally) defines other project-wide make variables. It includes the core `$(IDF_PATH)/make/project.mk` makefile which implements the rest of the ESP-IDF build system.
- “sdkconfig” project configuration file. This file is created/updated when “make menuconfig” runs, and holds configuration for all of the components in the project (including esp-idf itself). The “sdkconfig” file may or may not be added to the source control system of the project.
- Optional “components” directory contains components that are part of the project. A project does not have to contain custom components of this kind, but it can be useful for structuring reusable code or including third party components that aren’t part of ESP-IDF.
- “main” directory is a special “pseudo-component” that contains source code for the project itself. “main” is a default name, the Makefile variable `COMPONENT_DIRS` includes this component but you can modify this variable (or set `EXTRA_COMPONENT_DIRS`) to look for components in other places.
- “build” directory is where build output is created. After the make process is run, this directory will contain interim object files and libraries as well as final binary output files. This directory is usually not added to source control or distributed with the project source code.

Component directories contain a component makefile - `component.mk`. This may contain variable definitions to control the build process of the component, and its integration into the overall project. See [Component Makefiles](#) for more details.

Each component may also include a `Kconfig` file defining the *component configuration* options that can be set via the project configuration. Some components may also include `Kconfig.projbuild` and `Makefile.projbuild` files, which are special files for *overriding parts of the project*.

## Project Makefiles

Each project has a single Makefile that contains build settings for the entire project. By default, the project Makefile can be quite minimal.

### Minimal Example Makefile

```

PROJECT_NAME := myProject

include $(IDF_PATH)/make/project.mk

```

## Mandatory Project Variables

- `PROJECT_NAME`: Name of the project. Binary output files will use this name - ie `myProject.bin`, `myProject.elf`.

## Optional Project Variables

These variables all have default values that can be overridden for custom behaviour. Look in `make/project.mk` for all of the implementation details.

- `PROJECT_PATH`: Top-level project directory. Defaults to the directory containing the Makefile. Many other project variables are based on this variable. The project path cannot contain spaces.
- `BUILD_DIR_BASE`: The build directory for all objects/libraries/binaries. Defaults to `$(PROJECT_PATH)/build`.
- `COMPONENT_DIRS`: Directories to search for components. Defaults to `$(IDF_PATH)/components`, `$(PROJECT_PATH)/components`, `$(PROJECT_PATH)/main` and `EXTRA_COMPONENT_DIRS`. Override this variable if you don't want to search for components in these places.
- `EXTRA_COMPONENT_DIRS`: Optional list of additional directories to search for components.
- `COMPONENTS`: A list of component names to build into the project. Defaults to all components found in the `COMPONENT_DIRS` directories.

Any paths in these Makefile variables should be absolute paths. You can convert relative paths using `$(PROJECT_PATH)/xxx`, `$(IDF_PATH)/xxx`, or use the Make function `$(abspath xxx)`.

These variables should all be set before the line `include $(IDF_PATH)/make/project.mk` in the Makefile.

## Component Makefiles

Each project contains one or more components, which can either be part of `esp-idf` or added from other component directories.

A component is any directory that contains a `component.mk` file.

## Searching for Components

The list of directories in `COMPONENT_DIRS` is searched for the project's components. Directories in this list can either be components themselves (ie they contain a `component.mk` file), or they can be top-level directories whose subdirectories are components.

Running the `make list-components` target dumps many of these variables and can help debug the discovery of component directories.

## Multiple components with the same name

When `esp-idf` is collecting all the components to compile, it will do this in the order specified by `COMPONENT_DIRS`; by default, this means the `idf` components first, the project components second and optionally the components in `EXTRA_COMPONENT_DIRS` last. If two or more of these directories contain component subdirectories with the same name, the component in the last place searched is used. This allows, for example, overriding `esp-idf` components with a modified version by simply copying the component from the `esp-idf` component directory to the project component tree and then modifying it there. If used in this way, the `esp-idf` directory itself can remain untouched.

## Minimal Component Makefile

The minimal `component.mk` file is an empty file(!). If the file is empty, the default component behaviour is set:

- All source files in the same directory as the makefile (`*.c`, `*.cpp`, `*.cc`, `*.S`) will be compiled into the component library
- A sub-directory “include” will be added to the global include search path for all other components.
- The component library will be linked into the project app.

See *example component makefiles* for more complete component makefile examples.

Note that there is a difference between an empty `component.mk` file (which invokes default component build behaviour) and no `component.mk` file (which means no default component build behaviour will occur.) It is possible for a component to have no *component.mk* file, if it only contains other files which influence the project configuration or build process.

## Preset Component Variables

The following component-specific variables are available for use inside `component.mk`, but should not be modified:

- `COMPONENT_PATH`: The component directory. Evaluates to the absolute path of the directory containing `component.mk`. The component path cannot contain spaces.
- `COMPONENT_NAME`: Name of the component. Defaults to the name of the component directory.
- `COMPONENT_BUILD_DIR`: The component build directory. Evaluates to the absolute path of a directory inside `$(BUILD_DIR_BASE)` where this component’s source files are to be built. This is also the Current Working Directory any time the component is being built, so relative paths in make targets, etc. will be relative to this directory.
- `COMPONENT_LIBRARY`: Name of the static library file (relative to the component build directory) that will be built for this component. Defaults to `$(COMPONENT_NAME).a`.

The following variables are set at the project level, but exported for use in the component build:

- `PROJECT_NAME`: Name of the project, as set in project Makefile
- `PROJECT_PATH`: Absolute path of the project directory containing the project Makefile.
- `COMPONENTS`: Name of all components that are included in this build.
- `CONFIG_*`: Each value in the project configuration has a corresponding variable available in make. All names begin with `CONFIG_`.
- `CC`, `LD`, `AR`, `OBJCOPY`: Full paths to each tool from the gcc xtensa cross-toolchain.
- `HOSTCC`, `HOSTLD`, `HOSTAR`: Full names of each tool from the host native toolchain.
- `IDF_VER`: Git version of ESP-IDF (produced by `git describe`)

If you modify any of these variables inside `component.mk` then this will not prevent other components from building but it may make your component hard to build and/or debug.

## Optional Project-Wide Component Variables

The following variables can be set inside `component.mk` to control build settings across the entire project:

- `COMPONENT_ADD_INCLUDEDIRS`: Paths, relative to the component directory, which will be added to the include search path for all components in the project. Defaults to `include` if not overridden. If an include directory is only needed to compile this specific component, add it to `COMPONENT_PRIV_INCLUDEDIRS` instead.
- `COMPONENT_ADD_LDFLAGS`: Add linker arguments to the `LDFLAGS` for the app executable. Defaults to `-l$(COMPONENT_NAME)`. If adding pre-compiled libraries to this directory, add them as absolute paths - ie `$(COMPONENT_PATH)/libwhatever.a`
- `COMPONENT_DEPENDS`: Optional list of component names that should be compiled before this component. This is not necessary for link-time dependencies, because all component include directories are available at all times. It is necessary if one component generates an include file which you then want to include in another component. Most components do not need to set this variable.
- `COMPONENT_ADD_LINKER_DEPS`: Optional list of component-relative paths to files which should trigger a re-link of the ELF file if they change. Typically used for linker script files and binary libraries. Most components do not need to set this variable.

The following variable only works for components that are part of `esp-idf` itself:

- `COMPONENT_SUBMODULES`: Optional list of git submodule paths (relative to `COMPONENT_PATH`) used by the component. These will be checked (and initialised if necessary) by the build process. This variable is ignored if the component is outside the `IDF_PATH` directory.

## Optional Component-Specific Variables

The following variables can be set inside `component.mk` to control the build of that component:

- `COMPONENT_PRIV_INCLUDEDIRS`: Directory paths, must be relative to the component directory, which will be added to the include search path for this component's source files only.
- `COMPONENT_EXTRA_INCLUDES`: Any extra include paths used when compiling the component's source files. These will be prefixed with `-I` and passed as-is to the compiler. Similar to the `COMPONENT_PRIV_INCLUDEDIRS` variable, except these paths are not expanded relative to the component directory.
- `COMPONENT_SRCDIRS`: Directory paths, must be relative to the component directory, which will be searched for source files (`*.cpp`, `*.c`, `*.S`). Defaults to `'`, ie the component directory itself. Override this to specify a different list of directories which contain source files.
- `COMPONENT_OBJS`: Object files to compile. Default value is a `.o` file for each source file that is found in `COMPONENT_SRCDIRS`. Overriding this list allows you to exclude source files in `COMPONENT_SRCDIRS` that would otherwise be compiled. See *Specifying source files*
- `COMPONENT_EXTRA_CLEAN`: Paths, relative to the component build directory, of any files that are generated using custom make rules in the `component.mk` file and which need to be removed as part of `make clean`. See *Source Code Generation* for an example.
- `COMPONENT_OWNBUILDTARGET` & `COMPONENT_OWNCLEANTARGET`: These targets allow you to fully override the default build behaviour for the component. See *Fully Overriding The Component Makefile* for more details.
- `COMPONENT_CONFIG_ONLY`: If set, this flag indicates that the component produces no built output at all (ie `COMPONENT_LIBRARY` is not built), and most other component variables are ignored. This flag is used for IDF internal components which contain only `KConfig.projbuild` and/or `Makefile.projbuild` files to configure the project, but no source files.

- **CFLAGS:** Flags passed to the C compiler. A default set of **CFLAGS** is defined based on project settings. Component-specific additions can be made via **CFLAGS +=**. It is also possible (although not recommended) to override this variable completely for a component.
- **CPPFLAGS:** Flags passed to the C preprocessor (used for `.c`, `.cpp` and `.S` files). A default set of **CPPFLAGS** is defined based on project settings. Component-specific additions can be made via **CPPFLAGS +=**. It is also possible (although not recommended) to override this variable completely for a component.
- **CXXFLAGS:** Flags passed to the C++ compiler. A default set of **CXXFLAGS** is defined based on project settings. Component-specific additions can be made via **CXXFLAGS +=**. It is also possible (although not recommended) to override this variable completely for a component.

To apply compilation flags to a single source file, you can add a variable override as a target, ie:

```
apps/dhcpserver.o: CFLAGS += -Wno-unused-variable
```

This can be useful if there is upstream code that emits warnings.

## Component Configuration

Each component can also have a **Kconfig** file, alongside `component.mk`. This contains configuration settings to add to the “make menuconfig” for this component.

These settings are found under the “Component Settings” menu when `menuconfig` is run.

To create a component **KConfig** file, it is easiest to start with one of the **KConfig** files distributed with `esp-idf`.

For an example, see [Adding conditional configuration](#).

## Preprocessor Definitions

ESP-IDF build systems adds the following C preprocessor definitions on the command line:

- **ESP\_PLATFORM** — Can be used to detect that build happens within ESP-IDF.
- **IDF\_VER** — Defined to a git version string. E.g. `v2.0` for a tagged release or `v1.0-275-g0efaa4f` for an arbitrary commit.

## Build Process Internals

### Top Level: Project Makefile

- “make” is always run from the project directory and the project makefile, typically named `Makefile`.
- The project makefile sets **PROJECT\_NAME** and optionally customises other *optional project variables*
- The project makefile includes `$(IDF_PATH)/make/project.mk` which contains the project-level Make logic.
- `project.mk` fills in default project-level make variables and includes make variables from the project configuration. If the generated makefile containing project configuration is out of date, then it is regenerated (via targets in `project_config.mk`) and then the make process restarts from the top.
- `project.mk` builds a list of components to build, based on the default component directories or a custom list of components set in *optional project variables*.

- Each component can set some *optional project-wide component variables*. These are included via generated makefiles named `component_project_vars.mk` - there is one per component. These generated makefiles are included into `project.mk`. If any are missing or out of date, they are regenerated (via a recursive make call to the component makefile) and then the make process restarts from the top.
- *Makefile.projbuild* files from components are included into the make process, to add extra targets or configuration.
- By default, the project makefile also generates top-level build & clean targets for each component and sets up *app* and *clean* targets to invoke all of these sub-targets.
- In order to compile each component, a recursive make is performed for the component makefile.

To better understand the project make process, have a read through the `project.mk` file itself.

## Second Level: Component Makefiles

- Each call to a component makefile goes via the `$(IDF_PATH)/make/component_wrapper.mk` wrapper makefile.
- This component wrapper includes all `component Makefile.componentbuild` files, making any recipes, variables etc in these files available to every component.
- The `component_wrapper.mk` is called with the current directory set to the component build directory, and the `COMPONENT_MAKEFILE` variable is set to the absolute path to `component.mk`.
- `component_wrapper.mk` sets default values for all *component variables*, then includes the *component.mk* file which can override or modify these.
- If `COMPONENT_OWNBUILDTARGET` and `COMPONENT_OWNCLEANTARGET` are not defined, default build and clean targets are created for the component's source files and the prerequisite `COMPONENT_LIBRARY` static library file.
- The `component_project_vars.mk` file has its own target in `component_wrapper.mk`, which is evaluated from `project.mk` if this file needs to be rebuilt due to changes in the component makefile or the project configuration.

To better understand the component make process, have a read through the `component_wrapper.mk` file and some of the `component.mk` files included with esp-idf.

## Running Make Non-Interactively

When running `make` in a situation where you don't want interactive prompts (for example: inside an IDE or an automated build system) append `BATCH_BUILD=1` to the make arguments (or set it as an environment variable).

Setting `BATCH_BUILD` implies the following:

- Verbose output (same as `V=1`, see below). If you don't want verbose output, also set `V=0`.
- If the project configuration is missing new configuration items (from new components or esp-idf updates) then the project use the default values, instead of prompting the user for each item.
- If the build system needs to invoke `menuconfig`, an error is printed and the build fails.

## Debugging The Make Process

Some tips for debugging the esp-idf build system:

- Appending `V=1` to the make arguments (or setting it as an environment variable) will cause make to echo all commands executed, and also each directory as it is entered for a sub-make.
- Running `make -w` will cause make to echo each directory as it is entered for a sub-make - same as `V=1` but without also echoing all commands.
- Running `make --trace` (possibly in addition to one of the above arguments) will print out every target as it is built, and the dependency which caused it to be built.
- Running `make -p` prints a (very verbose) summary of every generated target in each makefile.

For more debugging tips and general make information, see the *GNU Make Manual*.

## Warning On Undefined Variables

By default, the build process will print a warning if an undefined variable is referenced (like `$(DOES_NOT_EXIST)`). This can be useful to find errors in variable names.

If you don't want this behaviour, it can be disabled in menuconfig's top level menu under *SDK tool configuration*.

Note that this option doesn't trigger a warning if `ifdef` or `ifndef` are used in Makefiles.

## Overriding Parts of the Project

### Makefile.projbuild

For components that have build requirements that must be evaluated in the top-level project make pass, you can create a file called `Makefile.projbuild` in the component directory. This makefile is included when `project.mk` is evaluated.

For example, if your component needs to add to `CFLAGS` for the entire project (not just for its own source files) then you can set `CFLAGS +=` in `Makefile.projbuild`.

`Makefile.projbuild` files are used heavily inside esp-idf, for defining project-wide build features such as `esptool.py` command line arguments and the bootloader "special app".

Note that `Makefile.projbuild` isn't necessary for the most common component uses - such as adding include directories to the project, or `LD_FLAGS` to the final linking step. These values can be customised via the `component.mk` file itself. See *Optional Project-Wide Component Variables* for details.

Take care when setting variables or targets in this file. As the values are included into the top-level project makefile pass, they can influence or break functionality across all components!

### KConfig.projbuild

This is an equivalent to `Makefile.projbuild` for *component configuration* `KConfig` files. If you want to include configuration options at the top-level of menuconfig, rather than inside the "Component Configuration" sub-menu, then these can be defined in the `KConfig.projbuild` file alongside the `component.mk` file.

Take care when adding configuration values in this file, as they will be included across the entire project configuration. Where possible, it's generally better to create a `KConfig` file for *component configuration*.

## Makefile.componentbuild

For components that e.g. include tools to generate source files from other files, it is necessary to be able to add recipes, macros or variable definitions into the component build process of every components. This is done by having a `Makefile.componentbuild` in a component directory. This file gets included in `component_wrapper.mk`, before the `component.mk` of the component is included. As with the `Makefile.projbuild`, take care with these files: as they're included in each component build, a `Makefile.componentbuild` error may only show up when compiling an entirely different component.

## Configuration-Only Components

Some special components which contain no source files, only `Kconfig.projbuild` and `Makefile.projbuild`, may set the flag `COMPONENT_CONFIG_ONLY` in the `component.mk` file. If this flag is set, most other component variables are ignored and no build step is run for the component.

## Example Component Makefiles

Because the build environment tries to set reasonable defaults that will work most of the time, `component.mk` can be very small or even empty (see [Minimal Component Makefile](#)). However, overriding *component variables* is usually required for some functionality.

Here are some more advanced examples of `component.mk` makefiles:

### Adding source directories

By default, sub-directories are ignored. If your project has sources in sub-directories instead of in the root of the component then you can tell that to the build system by setting `COMPONENT_SRCDIRS`:

```
COMPONENT_SRCDIRS := src1 src2
```

This will compile all source files in the `src1/` and `src2/` sub-directories instead.

### Specifying source files

The standard `component.mk` logic adds all `.S` and `.c` files in the source directories as sources to be compiled unconditionally. It is possible to circumvent that logic and hard-code the objects to be compiled by manually setting the `COMPONENT_OBJS` variable to the name of the objects that need to be generated:

```
COMPONENT_OBJS := file1.o file2.o thing/filea.o thing/fileb.o anotherthing/main.o
COMPONENT_SRCDIRS := . thing anotherthing
```

Note that `COMPONENT_SRCDIRS` must be set as well.

### Adding conditional configuration

The configuration system can be used to conditionally compile some files depending on the options selected in `make menuconfig`. For this, ESP-IDF has the `compile_only_if` and `compile_only_if_not` macros:

`Kconfig`:

```
config FOO_ENABLE_BAR
    bool "Enable the BAR feature."
    help
        This enables the BAR feature of the FOO component.
```

component.mk:

```
$(call compile_only_if,$(CONFIG_FOO_ENABLE_BAR),bar.o)
```

As can be seen in the example, the `compile_only_if` macro takes a condition and a list of object files as parameters. If the condition is true (in this case: if the BAR feature is enabled in menuconfig) the object files (in this case: `bar.o`) will always be compiled. The opposite goes as well: if the condition is not true, `bar.o` will never be compiled. `compile_only_if_not` does the opposite: compile if the condition is false, not compile if the condition is true.

This can also be used to select or stub out an implementation, as such:

Kconfig:

```
config ENABLE_LCD_OUTPUT
    bool "Enable LCD output."
    help
        Select this if your board has a LCD.

config ENABLE_LCD_CONSOLE
    bool "Output console text to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output debugging output to the lcd

config ENABLE_LCD_PLOT
    bool "Output temperature plots to LCD"
    depends on ENABLE_LCD_OUTPUT
    help
        Select this to output temperature plots
```

component.mk:

```
# If LCD is enabled, compile interface to it, otherwise compile dummy interface
$(call compile_only_if,$(CONFIG_ENABLE_LCD_OUTPUT),lcd-real.o lcd-spi.o)
$(call compile_only_if_not,$(CONFIG_ENABLE_LCD_OUTPUT),lcd-dummy.o)

#We need font if either console or plot is enabled
$(call compile_only_if,$(or $(CONFIG_ENABLE_LCD_CONSOLE),$(CONFIG_ENABLE_LCD_PLOT)),_
↪font.o)
```

Note the use of the Make ‘or’ function to include the font file. Other substitution functions, like ‘and’ and ‘if’ will also work here. Variables that do not come from menuconfig can also be used: ESP-IDF uses the default Make policy of judging a variable which is empty or contains only whitespace to be false while a variable with any non-whitespace in it is true.

(Note: Older versions of this document advised conditionally adding object file names to `COMPONENT_OBJS`. While this still is possible, this will only work when all object files for a component are named explicitly, and will not clean up deselected object files in a `make clean` pass.)

## Source Code Generation

Some components will have a situation where a source file isn't supplied with the component itself but has to be generated from another file. Say our component has a header file that consists of the converted binary data of a BMP file, converted using a hypothetical tool called `bmp2h`. The header file is then included in as C source file called `graphics_lib.c`:

```
COMPONENT_EXTRA_CLEAN := logo.h

graphics_lib.o: logo.h

logo.h: $(COMPONENT_PATH)/logo.bmp
    bmp2h -i $^ -o $@
```

In this example, `graphics_lib.o` and `logo.h` will be generated in the current directory (the build directory) while `logo.bmp` comes with the component and resides under the component path. Because `logo.h` is a generated file, it needs to be cleaned when `make clean` is called which why it is added to the `COMPONENT_EXTRA_CLEAN` variable.

## Cosmetic Improvements

Because `logo.h` is a generated file, it needs to be cleaned when `make clean` is called which why it is added to the `COMPONENT_EXTRA_CLEAN` variable.

Adding `logo.h` to the `graphics_lib.o` dependencies causes it to be generated before `graphics_lib.c` is compiled.

If a source file in another component included `logo.h`, then this component's name would have to be added to the other component's `COMPONENT_DEPENDS` list to ensure that the components were built in-order.

## Embedding Binary Data

Sometimes you have a file with some binary or text data that you'd like to make available to your component - but you don't want to reformat the file as C source.

You can set a variable `COMPONENT_EMBED_FILES` in `component.mk`, giving the names of the files to embed in this way:

```
COMPONENT_EMBED_FILES := server_root_cert.der
```

Or if the file is a string, you can use the variable `COMPONENT_EMBED_TXTFILES`. This will embed the contents of the text file as a null-terminated string:

```
COMPONENT_EMBED_TXTFILES := server_root_cert.pem
```

The file's contents will be added to the `.rodata` section in flash, and are available via symbol names as follows:

```
extern const uint8_t server_root_cert_pem_start[] asm("_binary_server_root_cert_pem_
↪start");
extern const uint8_t server_root_cert_pem_end[]    asm("_binary_server_root_cert_pem_
↪end");
```

The names are generated from the full name of the file, as given in `COMPONENT_EMBED_FILES`. Characters `/`, `.`, etc. are replaced with underscores. The `_binary` prefix in the symbol name is added by `objcopy` and is the same for both text and binary files.

For an example of using this technique, see [protocols/https\\_request](#) - the certificate file contents are loaded from the text `.pem` file at compile time.

## Fully Overriding The Component Makefile

Obviously, there are cases where all these recipes are insufficient for a certain component, for example when the component is basically a wrapper around another third-party component not originally intended to be compiled under this build system. In that case, it's possible to forego the esp-idf build system entirely by setting `COMPONENT_OWNBUILDTARGET` and possibly `COMPONENT_OWNCLEANTARGET` and defining your own targets named `build` and `clean` in component `.mk` target. The build target can do anything as long as it creates `$(COMPONENT_LIBRARY)` for the project make process to link into the app binary.

(Actually, even this is not strictly necessary - if the `COMPONENT_ADD_LDFLAGS` variable is overridden then the component can instruct the linker to link other binaries instead.)

## Custom sdkconfig defaults

For example projects or other projects where you don't want to specify a full `sdkconfig` configuration, but you do want to override some key values from the esp-idf defaults, it is possible to create a file `sdkconfig.defaults` in the project directory. This file will be used when running `make defconfig`, or creating a new config from scratch.

To override the name of this file, set the `SDKCONFIG_DEFAULTS` environment variable.

## Save flash arguments

There're some scenarios that we want to flash the target board without IDF. For this case we want to save the built binaries, `esptool.py` and `esptool write_flash` arguments. It's simple to write a script to save binaries and `esptool.py`. We can use command `make print_flash_cmd`, it will print the flash arguments:

```
--flash_mode dio --flash_freq 40m --flash_size detect 0x1000 bootloader/bootloader.  
↪bin 0x10000 example_app.bin 0x8000 partition_table_unit_test_app.bin
```

Then use flash arguments as the arguemnts for `esptool write_flash` arguments:

```
python esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 921600 --before default_  
↪reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_  
↪size detect 0x1000 bootloader/bootloader.bin 0x10000 example_app.bin 0x8000_  
↪partition_table_unit_test_app.bin
```

## 4.2.3 Building the Bootloader

The bootloader is built by default as part of “make all”, or can be built standalone via “make bootloader-clean”. There is also “make bootloader-list-components” to see the components included in the bootloader build.

The component in IDF `components/bootloader` is special, as the second stage bootloader is a separate `.ELF` and `.BIN` file to the main project. However it shares its configuration and build directory with the main project.

This is accomplished by adding a subproject under `components/bootloader/subproject`. This subproject has its own Makefile, but it expects to be called from the project's own Makefile via some glue in the `components/bootloader/Makefile.projectbuild` file. See these files for more details.

## 4.3 Deep Sleep Wake Stubs

ESP32 supports running a “deep sleep wake stub” when coming out of deep sleep. This function runs immediately as soon as the chip wakes up - before any normal initialisation, bootloader, or ESP-IDF code has run. After the wake stub runs, the SoC can go back to sleep or continue to start ESP-IDF normally.

Deep sleep wake stub code is loaded into “RTC Fast Memory” and any data which it uses must also be loaded into RTC memory. RTC memory regions hold their contents during deep sleep.

### 4.3.1 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.

### 4.3.2 Implementing A Stub

The wake stub in esp-idf is called `esp_wake_deep_sleep()`. This function runs whenever the SoC wakes from deep sleep. There is a default version of this function provided in esp-idf, but the default function is weak-linked so if your app contains a function named `esp_wake_deep_sleep()` then this will override the default.

If supplying a custom wake stub, the first thing it does should be to call `esp_default_wake_deep_sleep()`.

It is not necessary to implement `esp_wake_deep_sleep()` in your app in order to use deep sleep. It is only necessary if you want to have special behaviour immediately on wake.

If you want to swap between different deep sleep stubs at runtime, it is also possible to do this by calling the `esp_set_deep_sleep_wake_stub()` function. This is not necessary if you only use the default `esp_wake_deep_sleep()` function.

All of these functions are declared in the `esp_deepsleep.h` header under `components/esp32`.

### 4.3.3 Loading Code Into RTC Memory

Wake stub code must be resident in RTC Fast Memory. This can be done in one of two ways.

The first way is to use the `RTC_IRAM_ATTR` attribute to place a function into RTC memory:

```
void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    // Add additional functionality here
}
```

The second way is to place the function into any source file whose name starts with `rtc_wake_stub`. Files names `rtc_wake_stub*` have their contents automatically put into RTC memory by the linker.

The first way is simpler for very short and simple code, or for source files where you want to mix “normal” and “RTC” code. The second way is simpler when you want to write longer pieces of code for RTC memory.

### 4.3.4 Loading Data Into RTC Memory

Data used by stub code must be resident in RTC Slow Memory. This memory 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 slow memory:

```
RTC_DATA_ATTR int wake_count;

void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    static RTC_RODATA_ATTR const char fmt_str[] = "Wake count %d\n";
    ets_printf(fmt_str, wake_count++);
}
```

Unfortunately, any string constants used in this way must be declared as arrays and marked with `RTC_RODATA_ATTR`, as shown in the example above.

The second way is to place the data into any source file whose name starts with `rtc_wake_stub`.

For example, the equivalent example in `rtc_wake_stub_counter.c`:

```
int wake_count;

void RTC_IRAM_ATTR esp_wake_deep_sleep(void) {
    esp_default_wake_deep_sleep();
    ets_printf("Wake count %d\n", wake_count++);
}
```

The second way is a better option if you need to use strings, or write other more complex code.

## 4.4 ESP32 Core Dump

### 4.4.1 Overview

ESP-IDF provides support to generate core dumps on unrecoverable software errors. This useful technique allows post-mortem analysis of software state at the moment of failure. Upon the crash system enters panic state, prints some information and halts or reboots depending configuration. User can choose to generate core dump in order to analyse the reason of failure on PC later on. Core dump contains snapshots of all tasks in the system at the moment of failure. Snapshots include tasks control blocks (TCB) and stacks. So it is possible to find out what task, at what instruction (line of code) and what callstack of that task lead to the crash. ESP-IDF provides special script `espcoredump.py` to help users to retrieve and analyse core dumps. This tool provides two commands for core dumps analysis:

- `info_corefile` - prints crashed task’s registers, callstack, list of available tasks in the system, memory regions and contents of memory stored in core dump (TCBs and stacks)

- `dbg_corefile` - creates core dump ELF file and runs GDB debug session with this file. User can examine memory, variables and tasks states manually. Note that since not all memory is saved in core dump only values of variables allocated on stack will be meaningful

## 4.4.2 Configuration

There are a number of core dump related configuration options which user can choose in configuration menu of the application (*make menuconfig*).

1. Core dump data destination (*Components -> ESP32-specific config -> Core dump destination*):
  - Disable core dump generation
  - Save core dump to flash
  - Print core dump to UART
2. Logging level of core dump module (*Components -> ESP32-specific config -> Core dump module logging level*). Value is a number from 0 (no output) to 5 (most verbose).
3. Delay before core dump will be printed to UART (*Components -> ESP32-specific config -> Core dump print to UART delay*). Value is in ms.

## 4.4.3 Save core dump to flash

When this option is selected core dumps are saved to special partition on flash. When using default partition table files which are provided with ESP-IDF it automatically allocates necessary space on flash, But if user wants to use its own layout file together with core dump feature it should define separate partition for core dump as it is shown below:

```
# Name,   Type, SubType, Offset, Size
# Note: if you change the phy_init or app partition offset, make sure to change the
#       ↪offset in Kconfig.projbuild
nvs,     data, nvs,     0x9000, 0x6000
phy_init, data, phy,     0xf000, 0x1000
factory, app,  factory, 0x10000, 1M
coredump, data, coredump,, 64K
```

There are no special requirements for partition name. It can be chosen according to the user application needs, but partition type should be 'data' and sub-type should be 'coredump'. Also when choosing partition size note that core dump data structure introduces constant overhead of 20 bytes and per-task overhead of 12 bytes. This overhead does not include size of TCB and stack for every task. So partition size should be at least 20 + max tasks number x (12 + TCB size + max task stack size) bytes.

The example of generic command to analyze core dump from flash is: `espcoredump.py -p </path/to/serial/port> info_corefile </path/to/program/elf/file>` or `espcoredump.py -p </path/to/serial/port> dbg_corefile </path/to/program/elf/file>`

## 4.4.4 Print core dump to UART

When this option is selected base64-encoded core dumps are printed on UART upon system panic. In this case user should save core dump text body to some file manually and then run the following command: `espcoredump.py info_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>` or `espcoredump.py dbg_corefile -t b64 -c </path/to/saved/base64/text> </path/to/program/elf/file>`

Base64-encoded body of core dump will be between the following header and footer:

```
===== CORE DUMP START =====  
<body of base64-encoded core dump, save it to file on disk>  
===== CORE DUMP END =====
```

## 4.4.5 Running ‘`espcoredump.py`’

Generic command syntax:

`espcoredump.py [options] command [args]`

### Script Options

- `-chip,-c {auto,esp32}`. Target chip type. Supported values are *auto* and *esp32*.
- `-port,-p PORT`. Serial port device.
- `-baud,-b BAUD`. Serial port baud rate used when flashing/reading.

### Commands

- `info_corefile`. Retrieve core dump and print useful info.
- `dbg_corefile`. Retrieve core dump and start GDB session with it.

### Command Arguments

- `-gdb,-g GDB`. Path to gdb to use for data retrieval.
- `-core,-c CORE`. Path to core dump file to use (if skipped core dump will be read from flash).
- `-core-format,-t CORE_FORMAT`. Specifies that file passed with “-c” is an ELF (“elf”), dumped raw binary (“raw”) or base64-encoded (“b64”) format.
- `-off,-o OFF`. Offset of coredump partition in flash (type “make partition\_table” to see it).
- `-save-core,-s SAVE_CORE`. Save core to file. Otherwise temporary core file will be deleted. Ignored with “-c”.
- `-print-mem,-m` Print memory dump. Used only with “info\_corefile”.

## 4.5 Flash Encryption

Flash Encryption is a feature for encrypting the contents of the ESP32’s attached SPI flash. When flash encryption is enabled, physical readout of the SPI flash is not sufficient to recover most flash contents.

Flash Encryption is separate from the *Secure Boot* feature, and you can use flash encryption without enabling secure boot. However, **for a secure environment both should be used simultaneously**. In absence of secure boot, additional configuration needs to be performed to ensure effectiveness of flash encryption. See *Securing Flash Encryption* for more details.

When using any non-default configuration in production, additional steps may also be needed to ensure effectiveness of flash encryption. See *Securing Flash Encryption* for more details.

**IMPORTANT: Enabling flash encryption limits your options for further updates of your ESP32. Make sure to read this document (including [:ref:‘flash-encryption-limitations’](#)) and understand the implications of enabling flash encryption.**

### 4.5.1 Background

- The contents of the flash are encrypted using AES with a 256 bit key. The flash encryption key is stored in efuse internal to the chip, and is (by default) protected from software access.
- Flash access is transparent via the flash cache mapping feature of ESP32 - any flash regions which are mapped to the address space will be transparently decrypted when read.
- Encryption is applied by flashing the ESP32 with plaintext data, and (if encryption is enabled) the bootloader encrypts the data in place on first boot.
- Not all of the flash is encrypted. The following kinds of flash data are encrypted:
  - Bootloader
  - Secure boot bootloader digest (if secure boot is enabled)
  - Partition Table
  - All “app” type partitions
  - Any partition marked with the “encrypt” flag in the partition table

It may be desirable for some data partitions to remain unencrypted for ease of access, or to use flash-friendly update algorithms that are ineffective if the data is encrypted. “NVS” partitions for non-volatile storage cannot be encrypted.

- The flash encryption key is stored in efuse key block 1, internal to the ESP32 chip. By default, this key is read-and write-protected so software cannot access it or change it.
- The *flash encryption algorithm* is AES-256, where the key is “tweaked” with the offset address of each 32 byte block of flash. This means every 32 byte block (two consecutive 16 byte AES blocks) is encrypted with a unique key derived from the flash encryption key.
- Although software running on the chip can transparently decrypt flash contents, by default it is made impossible for the UART bootloader to decrypt (or encrypt) data when flash encryption is enabled.
- If flash encryption may be enabled, the programmer must take certain precautions when writing code that *uses encrypted flash*.

### 4.5.2 Flash Encryption Initialisation

This is the default (and recommended) flash encryption initialisation process. It is possible to customise this process for development or other purposes, see *Flash Encryption Advanced Features* for details.

**IMPORTANT: Once flash encryption is enabled on first boot, the hardware allows a maximum of 3 subsequent flash updates via serial re-flashing.** A special procedure (documented in *Serial Flashing*) must be followed to perform these updates.

- If secure boot is enabled, no physical re-flashes are possible.
- OTA updates can be used to update flash content without counting towards this limit.
- When enabling flash encryption in development, use a *pregenerated flash encryption key* to allow physically re-flashing an unlimited number of times with pre-encrypted data.\*\*

Process to enable flash encryption:

- The bootloader must be compiled with flash encryption support enabled. In `make menuconfig`, navigate to “Security Features” and select “Yes” for “Enable flash encryption on boot”.
- If enabling Secure Boot at the same time, it is best to simultaneously select those options now. Read the *Secure Boot* documentation first.

- Build and flash the bootloader, partition table and factory app image as normal. These partitions are initially written to the flash unencrypted.
- On first boot, the bootloader sees *FLASH\_CRYPT\_CNT* efuse is set to 0 (factory default) so it generates a flash encryption key using the hardware random number generator. This key is stored in efuse. The key is read and write protected against further software access.
- All of the encrypted partitions are then encrypted in-place by the bootloader. Encrypting in-place can take some time (up to a minute for large partitions.)

**IMPORTANT: Do not interrupt power to the ESP32 while the first boot encryption pass is running. If power is interrupted, the flash contents will be corrupted and require flashing with unencrypted data again. A reflash like this will not count towards the flashing limit.**

- Once flashing is complete. efuses are blown (by default) to disable encrypted flash access while the UART bootloader is running. See *Enabling UART Bootloader Encryption/Decryption* for advanced details.
- The *FLASH\_CRYPT\_CONFIG* efuse is also burned to the maximum value (0xF) to maximise the number of key bits which are tweaked in the flash algorithm. See *Setting FLASH\_CRYPT\_CONFIG* for advanced details.
- Finally, the *FLASH\_CRYPT\_CNT* efuse is burned with the initial value 1. It is this efuse which activates the transparent flash encryption layer, and limits the number of subsequent reflashes. See the *Updating Encrypted Flash* section for details about *FLASH\_CRYPT\_CNT* efuse.
- The bootloader resets itself to reboot from the newly encrypted flash.

### 4.5.3 Using Encrypted Flash

ESP32 app code can check if flash encryption is currently enabled by calling `esp_flash_encryption_enabled()`.

Once flash encryption is enabled, some care needs to be taken when accessing flash contents from code.

#### Scope of Flash Encryption

Whenever the *FLASH\_CRYPT\_CNT* efuse is set to a value with an odd number of bits set, all flash content which is accessed via the MMU's flash cache is transparently decrypted. This includes:

- Executable application code in flash (IROM).
- All read-only data stored in flash (DROM).
- Any data accessed via `esp_spi_flash_mmap()`.
- The software bootloader image when it is read by the ROM bootloader.

**IMPORTANT: The MMU flash cache unconditionally decrypts all data. Data which is stored unencrypted in the flash will be “transparently decrypted” via the flash cache and appear to software like random garbage.**

#### Reading Encrypted Flash

To read data without using a flash cache MMU mapping, we recommend using the partition read function `esp_partition_read()`. When using this function, data will only be decrypted when it is read from an encrypted partition. Other partitions will be read unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

Data which is read via other SPI read APIs are not decrypted:

- Data read via `esp_spi_flash_read()` is not decrypted

- Data read via ROM function `SPIRead()` is not decrypted (this function is not supported in esp-idf apps).
- Data stored using the Non-Volatile Storage (NVS) API is always stored and read decrypted.

## Writing Encrypted Flash

Where possible, we recommend using the partition write function `esp_partition_write`. When using this function, data will only be encrypted when writing to encrypted partitions. Data will be written to other partitions unencrypted. In this way, software can access encrypted and non-encrypted flash in the same way.

The `esp_spi_flash_write` function will write data when the `write_encrypted` parameter is set to true. Otherwise, data will be written unencrypted.

The ROM function `esp_rom_spiflash_write_encrypted` will write encrypted data to flash, the ROM function `SPIWrite` will write unencrypted to flash. (these function are not supported in esp-idf apps).

The minimum write size for unencrypted data is 4 bytes (and the alignment is 4 bytes). Because data is encrypted in blocks, the minimum write size for encrypted data is 16 bytes (and the alignment is 16 bytes.)

## 4.5.4 Updating Encrypted Flash

### OTA Updates

OTA updates to encrypted partitions will automatically write encrypted, as long as the `esp_partition_write` function is used.

### Serial Flashing

Provided secure boot is not used, the `FLASH_CRYPT_CNT` *efuse* allows the flash to be updated with new plaintext data via serial flashing (or other physical methods), up to 3 additional times.

The process involves flashing plaintext data, and then bumping the value of `FLASH_CRYPT_CNT` *efuse* which causes the bootloader to re-encrypt this data.

### Limited Updates

Only 4 serial flash update cycles of this kind are possible, including the initial encrypted flash.

After the fourth time encryption is enabled, `FLASH_CRYPT_CNT` *efuse* has the maximum value `0x7F` (7 bits set) and encryption is permanently enabled.

Using *OTA Updates* or *Reflashing via Pregenerated Flash Encryption Key* allows you to exceed this limit.

### Cautions With Serial Flashing

- When reflashing via serial, reflash every partition that was initially written with plaintext data (including bootloader). It is possible to skip app partitions which are not the “currently selected” OTA partition (these will not be re-encrypted unless a plaintext app image is found there.) However any partition marked with the “encrypt” flag will be unconditionally re-encrypted, meaning that any already encrypted data will be encrypted twice and corrupted.
  - Using `make flash` should flash all partitions which need to be flashed.

- If secure boot is enabled, you can't reflash via serial at all unless you used the "Reflashable" option for Secure Boot, pre-generated a key and burned it to the ESP32 (refer to *Secure Boot* docs.). In this case you can re-flash a plaintext secure boot digest and bootloader image at offset 0x0. It is necessary to re-flash this digest before flashing other plaintext data.

### Serial Re-Flashing Procedure

- Build the application as usual.
- Flash the device with plaintext data as usual (`make flash` or `esptool.py` commands.) Flash all previously encrypted partitions, including the bootloader (see previous section).
- At this point, the device will fail to boot (message is `flash read err, 1000`) because it expects to see an encrypted bootloader, but the bootloader is plaintext.
- Burn the `FLASH_CRYPT_CNT` *efuse* by running the command `espefuse.py burn_efuse FLASH_CRYPT_CNT`. `espefuse.py` will automatically increment the bit count by 1, which disables encryption.
- Reset the device and it will re-encrypt plaintext partitions, then burn the `FLASH_CRYPT_CNT` *efuse* again to re-enable encryption.

To prevent any further serial updates, see *Securing Flash Encryption*.

### Reflashing via Pregenerated Flash Encryption Key

It is possible to pregenerate a flash encryption key on the host computer and burn it into the ESP32's efuse key block. This allows data to be pre-encrypted on the host and flashed to the ESP32 without needing a plaintext flash update.

This is useful for development, because it removes the 4 time reflashing limit. It also allows reflashing with secure boot enabled, because the bootloader doesn't need to be reflashed each time.

**IMPORTANT** This method is intended to assist with development only, not for production devices. If pre-generating flash encryption for production, ensure the keys are generated from a high quality random number source and do not share the same flash encryption key across multiple devices.

### Pregenerating a Flash Encryption Key

Flash encryption keys are 32 bytes of random data. You can generate a random key with `espsecure.py`:

```
espsecure.py generate_flash_encryption_key my_flash_encryption_key.bin
```

(The randomness of this data is only as good as the OS and it's Python installation's random data source.)

Alternatively, if you're using *secure boot* and have a secure boot signing key then you can generate a deterministic SHA-256 digest of the secure boot private signing key and use this as the flash encryption key:

```
espsecure.py digest_private-key --keyfile secure_boot_signing_key.pem my_flash_
→ encryption_key.bin
```

(The same 32 bytes is used as the secure boot digest key if you enable *reflashable mode* for secure boot.)

Generating the flash encryption key from the secure boot signing key in this way means that you only need to store one key file. However this method is **not at all suitable** for production devices.

## Burning Flash Encryption Key

Once you have generated a flash encryption key, you need to burn it to the ESP32's efuse key block. **This must be done before first encrypted boot**, otherwise the ESP32 will generate a random key that software can't access or modify.

To burn a key to the device (one time only):

```
espefuse.py --port PORT burn_key flash_encryption my_flash_encryption_key.bin
```

## First Flash with pregenerated key

After flashing the key, follow the same steps as for default *Flash Encryption Initialisation* and flash a plaintext image for the first boot. The bootloader will enable flash encryption using the pre-burned key and encrypt all partitions.

## Reflashing with pregenerated key

After encryption is enabled on first boot, reflashing an encrypted image requires an additional manual step. This is where we pre-encrypt the data that we wish to update in flash.

Suppose that this is the normal command used to flash plaintext data:

```
esptool.py --port /dev/ttyUSB0 --baud 115200 write_flash 0x10000 build/my-app.bin
```

Binary app image `build/my-app.bin` is written to offset `0x10000`. This file name and offset need to be used to encrypt the data, as follows:

```
espsecure.py encrypt_flash_data --keyfile my_flash_encryption_key.bin --address_
↪0x10000 -o build/my-app-encrypted.bin build/my-app.bin
```

This example command will encrypts `my-app.bin` using the supplied key, and produce an encrypted file `my-app-encrypted.bin`. Be sure that the address argument matches the address where you plan to flash the binary.

Then, flash the encrypted binary with `esptool.py`:

```
esptool.py --port /dev/ttyUSB0 --baud 115200 write_flash 0x10000 build/my-app-
↪encrypted.bin
```

No further steps or efuse manipulation is necessary, because the data is already encrypted when we flash it.

## 4.5.5 Disabling Flash Encryption

If you've accidentally enabled flash encryption for some reason, the next flash of plaintext data will soft-brick the ESP32 (the device will reboot continuously, printing the error `flash read err, 1000`).

You can disable flash encryption again by writing *FLASH\_CRYPT\_CNT* efuse:

- First, run `make menuconfig` and uncheck "Enable flash encryption boot" under "Security Features".
- Exit `menuconfig` and save the new configuration.
- Run `make menuconfig` again and double-check you really disabled this option! *If this option is left enabled, the bootloader will immediately re-enable encryption when it boots.*
- Run `make flash` to build and flash a new bootloader and app, without flash encryption enabled.

- Run `espefuse.py` (in `components/esptool_py/esptool`) to disable the `FLASH_CRYPT_CNT` efuse:  
`espefuse.py burn_efuse FLASH_CRYPT_CNT`

Reset the ESP32 and flash encryption should be disabled, the bootloader will boot as normal.

## 4.5.6 Limitations of Flash Encryption

Flash Encryption prevents plaintext readout of the encrypted flash, to protect firmware against unauthorised readout and modification. It is important to understand the limitations of the flash encryption system:

- Flash encryption is only as strong as the key. For this reason, we recommend keys are generated on the device during first boot (default behavior). If generating keys off-device (see [Reflashing via Pregenerated Flash Encryption Key](#)), ensure proper procedure is followed.
- 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 (ie to tell if two devices are probably running the same firmware version).
- For the same reason, an attacker can always tell when a pair of adjacent 16 byte blocks (32 byte aligned) contain identical content. Keep this in mind if storing sensitive data on the flash, design your flash storage so this doesn't happen (using a counter byte or some other non-identical value every 16 bytes is sufficient).
- 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](#).

## 4.5.7 Securing Flash Encryption

In a production setting it's important to ensure that flash encryption cannot be temporarily disabled.

This is because if the [Secure Boot](#) feature is not enabled, or if Secure Boot is somehow bypassed by an attacker, then unauthorised code can be written to flash in plaintext. This code can then re-enable encryption and access encrypted data, making flash encryption ineffective.

This problem must be avoided by write-protecting `FLASH_CRYPT_CNT` efuse and thereby keeping flash encryption permanently enabled.

The simplest way to do this is to enable the configuration option `CONFIG_FLASH_ENCRYPTION_DISABLE_PLAINTEXT` (enabled by default if Secure Boot is enabled). This option causes `FLASH_CRYPT_CNT` efuse to be write protected during initial app startup, or during first boot when the bootloader enables flash encryption. This includes if an app with this option is OTA updated.

Alternatively, `FLASH_CRYPT_CNT` efuse can be write-protected using the serial bootloader:

```
espefuse.py --port PORT write_protect_efuse FLASH_CRYPT_CNT
```

A third option with more flexibility: the app can call `esp_flash_write_protect_crypt_cnt()` at a convenient time during its startup or provisioning process, or set the `FLASH_ENCRYPTION_DISABLE_PLAINTEXT` config option for this to happen automatically.

## 4.5.8 Flash Encryption Advanced Features

The following information is useful for advanced use of flash encryption:

## Encrypted Partition Flag

Some partitions are encrypted by default. Otherwise, it is possible to mark any partition as requiring encryption:

In the *partition table* description CSV files, there is a field for flags.

Usually left blank, if you write “encrypted” in this field then the partition will be marked as encrypted in the partition table, and data written here will be treated as encrypted (same as an app partition):

```
# Name, Type, SubType, Offset, Size, Flags
nvs, data, nvs, 0x9000, 0x6000
phy_init, data, phy, 0xf000, 0x1000
factory, app, factory, 0x10000, 1M
secret_data, 0x40, 0x01, 0x20000, 256K, encrypted
```

- None of the default partition tables include any encrypted data partitions.
- It is not necessary to mark “app” partitions as encrypted, they are always treated as encrypted.
- The “encrypted” flag does nothing if flash encryption is not enabled.
- It is possible to mark the optional `phy` partition with `phy_init` data as encrypted, if you wish to protect this data from physical access readout or modification.
- It is not possible to mark the `nvs` partition as encrypted.

## Enabling UART Bootloader Encryption/Decryption

By default, on first boot the flash encryption process will burn efuses `DISABLE_DL_ENCRYPT`, `DISABLE_DL_DECRYPT` and `DISABLE_DL_CACHE`:

- `DISABLE_DL_ENCRYPT` disables the flash encryption operations when running in UART bootloader boot mode.
- `DISABLE_DL_DECRYPT` disables transparent flash decryption when running in UART bootloader mode, even if `FLASH_CRYPT_CNT` efuse is set to enable it in normal operation.
- `DISABLE_DL_CACHE` disables the entire MMU flash cache when running in UART bootloader mode.

It is possible to burn only some of these efuses, and write-protect the rest (with unset value 0) before the first boot, in order to preserve them. For example:

```
esefuse.py --port PORT burn_efuse DISABLE_DL_DECRYPT
esefuse.py --port PORT write_protect_efuse DISABLE_DL_ENCRYPT
```

(Note that all 3 of these efuses are disabled via one write protect bit, so write protecting one will write protect all of them. For this reason, it’s necessary to set any bits before write-protecting.)

**IMPORTANT:** Write protecting these efuses to keep them unset is not currently very useful, as `esptool.py` does not support writing or reading encrypted flash.

**IMPORTANT:** If `DISABLE_DL_DECRYPT` is left unset (0) this effectively makes flash encryption useless, as an attacker with physical access can use UART bootloader mode (with custom stub code) to read out the flash contents.

## Setting FLASH\_CRYPT\_CONFIG

The `FLASH_CRYPT_CONFIG` efuse determines the number of bits in the flash encryption key which are “tweaked” with the block offset. See *Flash Encryption Algorithm* for details.

First boot of the bootloader always sets this value to the maximum `0xF`.

It is possible to write these efuse manually, and write protect it before first boot in order to select different tweak values. This is not recommended.

It is strongly recommended to never write protect `FLASH_CRYPT_CONFIG` when its value is zero. If this efuse is set to zero, no bits in the flash encryption key are tweaked and the flash encryption algorithm is equivalent to AES ECB mode.

## 4.5.9 Technical Details

The following sections provide some reference information about the operation of flash encryption.

### FLASH\_CRYPT\_CNT efuse

`FLASH_CRYPT_CNT` is a 7-bit efuse field which controls flash encryption. Flash encryption enables or disables based on the number of bits in this efuse which are set to “1”:

- When an even number of bits (0,2,4,6) are set: Flash encryption is disabled, any encrypted data cannot be decrypted.
  - If the bootloader was built with “Enable flash encryption on boot” then it will see this situation and immediately re-encrypt the flash wherever it finds unencrypted data. Once done, it sets another bit in the efuse to ‘1’ meaning an odd number of bits are now set.
    1. On first plaintext boot, bit count has brand new value 0 and bootloader changes it to bit count 1 (value 0x01) following encryption.
    2. After next plaintext flash update, bit count is manually updated to 2 (value 0x03). After re-encrypting the bootloader changes efuse bit count to 3 (value 0x07).
    3. After next plaintext flash, bit count is manually updated to 4 (value 0x0F). After re-encrypting the bootloader changes efuse bit count to 5 (value 0x1F).
    4. After final plaintext flash, bit count is manually updated to 6 (value 0x3F). After re-encrypting the bootloader changes efuse bit count to 7 (value 0x7F).
- When an odd number of bits (1,3,5,7) are set: Transparent reading of encrypted flash is enabled.
- To avoid use of `FLASH_CRYPT_CNT efuse` state to disable flash encryption, load unauthorised code, then re-enabled flash encryption, secure boot must be used or `FLASH_CRYPT_CNT efuse` must be write-protected.

### Flash Encryption Algorithm

- AES-256 operates on 16 byte blocks of data. The flash encryption engine encrypts and decrypts data in 32 byte blocks, two AES blocks in series.
- AES algorithm is used inverted in flash encryption, so the flash encryption “encrypt” operation is AES decrypt and the “decrypt” operation is AES encrypt. This is for performance reasons and does not alter the effectiveness of the algorithm.
- The main flash encryption key is stored in efuse (`BLOCK1`) and by default is protected from further writes or software readout.
- Each 32 byte block (two adjacent 16 byte AES blocks) is encrypted with a unique key. The key is derived from the main flash encryption key in efuse, XORed with the offset of this block in the flash (a “key tweak”).
- The specific tweak depends on the setting of `FLASH_CRYPT_CONFIG` efuse. This is a 4 bit efuse, where each bit enables XORing of a particular range of the key bits:
  - Bit 1, bits 0-66 of the key are XORed.

- Bit 2, bits 67-131 of the key are XORed.
- Bit 3, bits 132-194 of the key are XORed.
- Bit 4, bits 195-256 of the key are XORed.

It is recommended that `FLASH_CRYPT_CONFIG` is always left to set the default value `0xF`, so that all key bits are XORed with the block offset. See *Setting `FLASH_CRYPT_CONFIG`* for details.

- The high 19 bits of the block offset (bit 5 to bit 23) are XORed with the main flash encryption key. This range is chosen for two reasons: the maximum flash size is 16MB (24 bits), and each block is 32 bytes so the least significant 5 bits are always zero.
- There is a particular mapping from each of the 19 block offset bits to the 256 bits of the flash encryption key, to determine which bit is XORed with which. See the variable `_FLASH_ENCRYPTION_TWEAK_PATTERN` in the `espsecure.py` source code for the complete mapping.
- To see the full flash encryption algorithm implemented in Python, refer to the `_flash_encryption_operation()` function in the `espsecure.py` source code.

## 4.6 ESP-IDF FreeRTOS SMP Changes

### 4.6.1 Overview

The vanilla FreeRTOS is designed to run on a single core. However the ESP32 is dual core containing a Protocol CPU (known as **CPU 0** or **PRO\_CPU**) and an Application CPU (known as **CPU 1** or **APP\_CPU**). The two cores are identical in practice and share the same memory. This allows the two cores to run tasks interchangeably between them.

The ESP-IDF FreeRTOS is a modified version of vanilla FreeRTOS which supports symmetric multiprocessing (SMP). ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0. This guide outlines the major differences between vanilla FreeRTOS and ESP-IDF FreeRTOS. The API reference for vanilla FreeRTOS can be found via <http://www.freertos.org/a00106.html>

*Backported Features:* Although ESP-IDF FreeRTOS is based on the Xtensa port of FreeRTOS v8.2.0, a number of FreeRTOS v9.0.0 features have been backported to ESP-IDF.

*Tasks and Task Creation:* Use `xTaskCreatePinnedToCore()` or `xTaskCreateStaticPinnedToCore()` to create tasks in ESP-IDF FreeRTOS. The last parameter of the two functions is `xCoreID`. This parameter specifies which core the task is pinned to. Acceptable values are 0 for **PRO\_CPU**, 1 for **APP\_CPU**, or `tskNO_AFFINITY` which allows the task to run on both.

*Round Robin Scheduling:* The ESP-IDF FreeRTOS scheduler will skip tasks when implementing Round-Robin scheduling between multiple tasks in the Ready state that are of the same priority. To avoid this behavior, ensure that those tasks either enter a blocked state, or are distributed across a wider range of priorities.

*Scheduler Suspension:* Suspending the scheduler in ESP-IDF FreeRTOS will only affect the scheduler on the the calling core. In other words, calling `vTaskSuspendAll()` on **PRO\_CPU** will not prevent **APP\_CPU** from scheduling, and vice versa. Use critical sections or semaphores instead for simultaneous access protection.

*Tick Interrupt Synchronicity:* Tick interrupts of **PRO\_CPU** and **APP\_CPU** are not synchronized. Do not expect to use `vTaskDelay()` or `vTaskDelayUntil()` as an accurate method of synchronizing task execution between the two cores. Use a counting semaphore instead as their context switches are not tied to tick interrupts due to preemption.

*Critical Sections & Disabling Interrupts:* In ESP-IDF FreeRTOS, critical sections are implemented using mutexes. Entering critical sections involve taking a mutex, then disabling the scheduler and interrupts of the calling core. However the other core is left unaffected. If the other core attempts to take same mutex, it will spin until the calling core has released the mutex by exiting the critical section.

*Floating Point Arithmetic:* The ESP32 supports hardware acceleration of single precision floating point arithmetic (`float`). However the use of hardware acceleration leads to some behavioral restrictions in ESP-IDF FreeRTOS. Therefore, tasks that utilize `float` will automatically be pinned to a core if not done so already. Furthermore, `float` cannot be used in interrupt service routines.

*Task Deletion:* Task deletion behavior has been backported from FreeRTOS v9.0.0 and modified to be SMP compatible. Task memory will be freed immediately when `vTaskDelete()` is called to delete a task that is not currently running and not pinned to the other core. Otherwise, freeing of task memory will still be delegated to the Idle Task.

*Thread Local Storage Pointers & Deletion Callbacks:* ESP-IDF FreeRTOS has backported the Thread Local Storage Pointers (TLSP) feature. However the extra feature of Deletion Callbacks has been added. Deletion callbacks are called automatically during task deletion and are used to free memory pointed to by TLSP. Call `vTaskSetThreadLocalStoragePointerAndDelCallback()` to set TLSP and Deletion Callbacks.

*FreeRTOS Hooks:* Vanilla FreeRTOS Hooks were not designed for SMP. ESP-IDF provides its own Idle and Tick Hooks in addition to the Vanilla FreeRTOS hooks. For full details, see the ESP-IDF Hooks API Reference.

*Configuring ESP-IDF FreeRTOS:* Several aspects of ESP-IDF FreeRTOS can be configured using `menuconfig` such as running ESP-IDF in Unicore Mode, or configuring the number of Thread Local Storage Pointers each task will have.

## 4.6.2 Backported Features

The following features have been backported from FreeRTOS v9.0.0 to ESP-IDF.

### Static Allocation

This feature has been backported from FreeRTOS v9.0.0 to ESP-IDF. The `SUPPORT_STATIC_ALLOCATION` option must be enabled in `menuconfig` in order for static allocation functions to be available. Once enabled, the following functions can be called...

- `xTaskCreateStatic()` (see *Backporting Notes* below)
- `xQueueCreateStatic`
- `xSemaphoreCreateBinaryStatic`
- `xSemaphoreCreateCountingStatic`
- `xSemaphoreCreateMutexStatic`
- `xSemaphoreCreateRecursiveMutexStatic`
- `xTimerCreateStatic()` (see *Backporting Notes* below)
- `xEventGroupCreateStatic()`

### Other Features

- `vTaskSetThreadLocalStoragePointer()` (see *Backporting Notes* below)
- `pvTaskGetThreadLocalStoragePointer()` (see *Backporting Notes* below)
- `vTimerSetTimerID()`
- `xTimerGetPeriod()`
- `xTimerGetExpiryTime()`
- `pcQueueGetName()`

- `uxSemaphoreGetCount`

## Backporting Notes

- 1) `xTaskCreateStatic()` has been made SMP compatible in a similar fashion to `xTaskCreate()` (see *Tasks and Task Creation*). Therefore `xTaskCreateStaticPinnedToCore()` can also be called.
- 2) Although vanilla FreeRTOS allows the Timer feature's daemon task to be statically allocated, the daemon task is always dynamically allocated in ESP-IDF. Therefore `vApplicationGetTimerTaskMemory` **does not** need to be defined when using statically allocated timers in ESP-IDF FreeRTOS.
- 3) The Thread Local Storage Pointer feature has been modified in ESP-IDF FreeRTOS to include Deletion Callbacks (see *Thread Local Storage Pointers & Deletion Callbacks*). Therefore the function `vTaskSetThreadLocalStoragePointerAndDelCallback()` can also be called.

## 4.6.3 Tasks and Task Creation

Tasks in ESP-IDF FreeRTOS are designed to run on a particular core, therefore two new task creation functions have been added to ESP-IDF FreeRTOS by appending `PinnedToCore` to the names of the task creation functions in vanilla FreeRTOS. The vanilla FreeRTOS functions of `xTaskCreate()` and `xTaskCreateStatic()` have led to the addition of `xTaskCreatePinnedToCore()` and `xTaskCreateStaticPinnedToCore()` in ESP-IDF FreeRTOS (see *Backported Features*).

For more details see `freertos/task.c`

The ESP-IDF FreeRTOS task creation functions are nearly identical to their vanilla counterparts with the exception of the extra parameter known as `xCoreID`. This parameter specifies the core on which the task should run on and can be one of the following values.

- 0 pins the task to **PRO\_CPU**
- 1 pins the task to **APP\_CPU**
- `tskNO_AFFINITY` allows the task to be run on both CPUs

For example `xTaskCreatePinnedToCore(tsk_callback, "APP_CPU Task", 1000, NULL, 10, NULL, 1)` creates a task of priority 10 that is pinned to **APP\_CPU** with a stack size of 1000 bytes. It should be noted that the `uxStackDepth` parameter in vanilla FreeRTOS specifies a task's stack depth in terms of the number of words, whereas ESP-IDF FreeRTOS specifies the stack depth in terms of bytes.

Note that the vanilla FreeRTOS functions `xTaskCreate()` and `xTaskCreateStatic()` have been defined in ESP-IDF FreeRTOS as inline functions which call `xTaskCreatePinnedToCore()` and `xTaskCreateStaticPinnedToCore()` respectively with `tskNO_AFFINITY` as the `xCoreID` value.

Each Task Control Block (TCB) in ESP-IDF stores the `xCoreID` as a member. Hence when each core calls the scheduler to select a task to run, the `xCoreID` member will allow the scheduler to determine if a given task is permitted to run on the core that called it.

## 4.6.4 Scheduling

The vanilla FreeRTOS implements scheduling in the `vTaskSwitchContext()` function. This function is responsible for selecting the highest priority task to run from a list of tasks in the Ready state known as the Ready Tasks List (described in the next section). In ESP-IDF FreeRTOS, each core will call `vTaskSwitchContext()` independently to select a task to run from the Ready Tasks List which is shared between both cores. There are several differences in scheduling behavior between vanilla and ESP-IDF FreeRTOS such as differences in Round Robin scheduling, scheduler suspension, and tick interrupt synchronicity.

## Round Robin Scheduling

Given multiple tasks in the Ready state and of the same priority, vanilla FreeRTOS implements Round Robin scheduling between each task. This will result in running those tasks in turn each time the scheduler is called (e.g. every tick interrupt). On the other hand, the ESP-IDF FreeRTOS scheduler may skip tasks when Round Robin scheduling multiple Ready state tasks of the same priority.

The issue of skipping tasks during Round Robin scheduling arises from the way the Ready Tasks List is implemented in FreeRTOS. In vanilla FreeRTOS, `pxReadyTasksList` is used to store a list of tasks that are in the Ready state. The list is implemented as an array of length `configMAX_PRIORITIES` where each element of the array is a linked list. Each linked list is of type `List_t` and contains TCBs of tasks of the same priority that are in the Ready state. The following diagram illustrates the `pxReadyTasksList` structure.

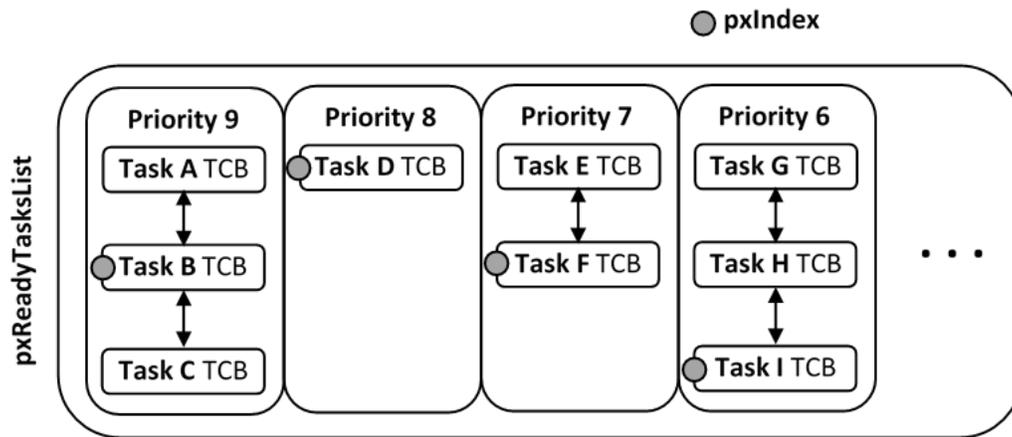


Fig. 1: Illustration of FreeRTOS Ready Task List Data Structure

Each linked list also contains a `pxIndex` which points to the last TCB returned when the list was queried. This index allows the `vTaskSwitchContext()` to start traversing the list at the TCB immediately after `pxIndex` hence implementing Round Robin Scheduling between tasks of the same priority.

In ESP-IDF FreeRTOS, the Ready Tasks List is shared between cores hence `pxReadyTasksList` will contain tasks pinned to different cores. When a core calls the scheduler, it is able to look at the `xCoreID` member of each TCB in the list to determine if a task is allowed to run on calling the core. The ESP-IDF FreeRTOS `pxReadyTasksList` is illustrated below.

Therefore when **PRO\_CPU** calls the scheduler, it will only consider the tasks in blue or purple. Whereas when **APP\_CPU** calls the scheduler, it will only consider the tasks in orange or purple.

Although each TCB has an `xCoreID` in ESP-IDF FreeRTOS, the linked list of each priority only has a single `pxIndex`. Therefore when the scheduler is called from a particular core and traverses the linked list, it will skip all TCBs pinned to the other core and point the `pxIndex` at the selected task. If the other core then calls the scheduler, it will traverse the linked list starting at the TCB immediately after `pxIndex`. Therefore, TCBs skipped on the previous scheduler call from the other core would not be considered on the current scheduler call. This issue is demonstrated in the following illustration.

Referring to the illustration above, assume that priority 9 is the highest priority, and none of the tasks in priority 9 will block hence will always be either in the running or Ready state.

- 1) **PRO\_CPU** calls the scheduler and selects Task A to run, hence moves `pxIndex` to point to Task A
- 2) **APP\_CPU** calls the scheduler and starts traversing from the task after `pxIndex` which is Task B. However Task B is not selected to run as it is not pinned to **APP\_CPU** hence it is skipped and Task C is selected instead. `pxIndex` now points to Task C

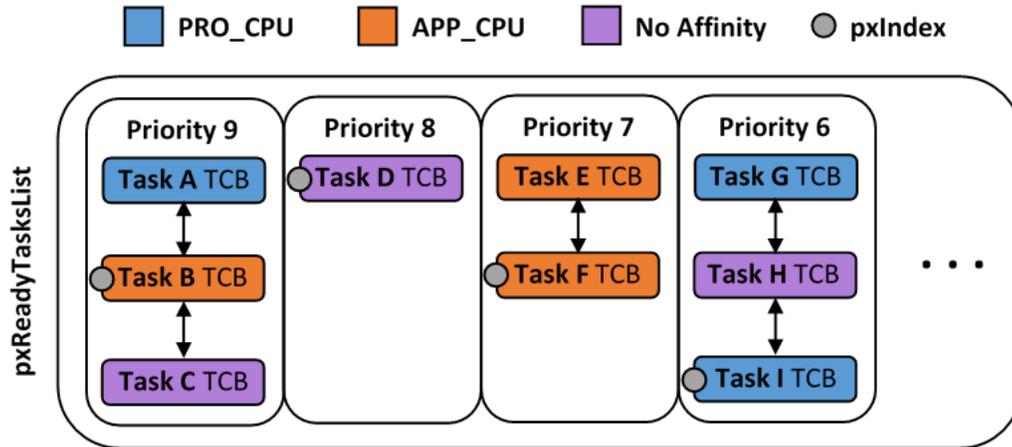


Fig. 2: Illustration of FreeRTOS Ready Task List Data Structure in ESP-IDF

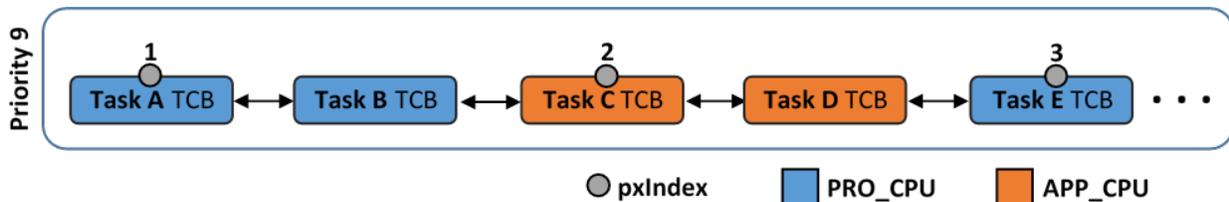


Fig. 3: Illustration of pxIndex behavior in ESP-IDF FreeRTOS

3) **PRO\_CPU** calls the scheduler and starts traversing from Task D. It skips Task D and selects Task E to run and points `pxIndex` to Task E. Notice that Task B isn't traversed because it was skipped the last time **APP\_CPU** called the scheduler to traverse the list.

4) The same situation with Task D will occur if **APP\_CPU** calls the scheduler again as `pxIndex` now points to Task E

One solution to the issue of task skipping is to ensure that every task will enter a blocked state so that they are removed from the Ready Task List. Another solution is to distribute tasks across multiple priorities such that a given priority will not be assigned multiple tasks that are pinned to different cores.

### Scheduler Suspension

In vanilla FreeRTOS, suspending the scheduler via `vTaskSuspendAll()` will prevent calls of `vTaskSwitchContext` from context switching until the scheduler has been resumed with `xTaskResumeAll()`. However servicing ISRs are still permitted. Therefore any changes in task states as a result from the current running task or ISRSs will not be executed until the scheduler is resumed. Scheduler suspension in vanilla FreeRTOS is a common protection method against simultaneous access of data shared between tasks, whilst still allowing ISRs to be serviced.

In ESP-IDF FreeRTOS, `xTaskResumeAll()` will only prevent calls of `vTaskSwitchContext()` from switching contexts on the core that called for the suspension. Hence if **PRO\_CPU** calls `vTaskSuspendAll()`, **APP\_CPU** will still be able to switch contexts. If data is shared between tasks that are pinned to different cores, scheduler suspension is **NOT** a valid method of protection against simultaneous access. Consider using critical sections (disables interrupts) or semaphores (does not disable interrupts) instead when protecting shared resources in ESP-IDF FreeRTOS.

In general, it's better to use other RTOS primitives like mutex semaphores to protect against data shared between tasks, rather than `vTaskSuspendAll()`.

### Tick Interrupt Synchronicity

In ESP-IDF FreeRTOS, tasks on different cores that unblock on the same tick count might not run at exactly the same time due to the scheduler calls from each core being independent, and the tick interrupts to each core being unsynchronized.

In vanilla FreeRTOS the tick interrupt triggers a call to `xTaskIncrementTick()` which is responsible for incrementing the tick counter, checking if tasks which have called `vTaskDelay()` have fulfilled their delay period, and moving those tasks from the Delayed Task List to the Ready Task List. The tick interrupt will then call the scheduler if a context switch is necessary.

In ESP-IDF FreeRTOS, delayed tasks are unblocked with reference to the tick interrupt on PRO\_CPU as PRO\_CPU is responsible for incrementing the shared tick count. However tick interrupts to each core might not be synchronized (same frequency but out of phase) hence when PRO\_CPU receives a tick interrupt, APP\_CPU might not have received it yet. Therefore if multiple tasks of the same priority are unblocked on the same tick count, the task pinned to PRO\_CPU will run immediately whereas the task pinned to APP\_CPU must wait until APP\_CPU receives its out of sync tick interrupt. Upon receiving the tick interrupt, APP\_CPU will then call for a context switch and finally switches contexts to the newly unblocked task.

Therefore, task delays should **NOT** be used as a method of synchronization between tasks in ESP-IDF FreeRTOS. Instead, consider using a counting semaphore to unblock multiple tasks at the same time.

### 4.6.5 Critical Sections & Disabling Interrupts

Vanilla FreeRTOS implements critical sections in `vTaskEnterCritical` which disables the scheduler and calls `portDISABLE_INTERRUPTS`. This prevents context switches and servicing of ISRs during a critical section. Therefore, critical sections are used as a valid protection method against simultaneous access in vanilla FreeRTOS.

On the other hand, the ESP32 has no hardware method for cores to disable each other's interrupts. Calling `portDISABLE_INTERRUPTS()` will have no effect on the interrupts of the other core. Therefore, disabling interrupts is **NOT** a valid protection method against simultaneous access to shared data as it leaves the other core free to access the data even if the current core has disabled its own interrupts.

For this reason, ESP-IDF FreeRTOS implements critical sections using mutexes, and calls to enter or exit a critical must provide a mutex that is associated with a shared resource requiring access protection. When entering a critical section in ESP-IDF FreeRTOS, the calling core will disable its scheduler and interrupts similar to the vanilla FreeRTOS implementation. However, the calling core will also take the mutex whilst the other core is left unaffected during the critical section. If the other core attempts to take the same mutex, it will spin until the mutex is released. Therefore, the ESP-IDF FreeRTOS implementation of critical sections allows a core to have protected access to a shared resource without disabling the other core. The other core will only be affected if it tries to concurrently access the same resource.

The ESP-IDF FreeRTOS critical section functions have been modified as follows...

- `taskENTER_CRITICAL(mux)`, `taskENTER_CRITICAL_ISR(mux)`, `portENTER_CRITICAL(mux)`, `portENTER_CRITICAL_ISR(mux)` are all macro defined to call `vTaskEnterCritical()`
- `taskEXIT_CRITICAL(mux)`, `taskEXIT_CRITICAL_ISR(mux)`, `portEXIT_CRITICAL(mux)`, `portEXIT_CRITICAL_ISR(mux)` are all macro defined to call `vTaskExitCritical()`

For more details see [freertos/include/freertos/portmacro.h](#) and [freertos/task.c](#)

It should be noted that when modifying vanilla FreeRTOS code to be ESP-IDF FreeRTOS compatible, it is trivial to modify the type of critical section called as they are all defined to call the same function. As long as the same mutex is provided upon entering and exiting, the type of call should not matter.

## 4.6.6 Floating Point Arithmetic

The ESP32 supports hardware acceleration of single precision floating point arithmetic (`float`) via Floating Point Units (FPU, also known as coprocessors) attached to each core. The use of the FPUs imposes some behavioral restrictions on ESP-IDF FreeRTOS.

ESP-IDF FreeRTOS implements Lazy Context Switching for FPUs. In other words, the state of a core's FPU registers are not immediately saved when a context switch occurs. Therefore, tasks that utilize `float` must be pinned to a particular core upon creation. If not, ESP-IDF FreeRTOS will automatically pin the task in question to whichever core the task was running on upon the task's first use of `float`. Likewise due to Lazy Context Switching, interrupt service routines must also not use `float`.

ESP32 does not support hardware acceleration for double precision floating point arithmetic (`double`). Instead `double` is implemented via software hence the behavioral restrictions with regards to `float` do not apply to `double`. Note that due to the lack of hardware acceleration, `double` operations may consume significantly larger amount of CPU time in comparison to `float`.

## 4.6.7 Task Deletion

FreeRTOS task deletion prior to v9.0.0 delegated the freeing of task memory entirely to the Idle Task. Currently, the freeing of task memory will occur immediately (within `vTaskDelete()`) if the task being deleted is not currently running or is not pinned to the other core (with respect to the core `vTaskDelete()` is called on). TLSP deletion callbacks will also run immediately if the same conditions are met.

However, calling `vTaskDelete()` to delete a task that is either currently running or pinned to the other core will still result in the freeing of memory being delegated to the Idle Task.

## 4.6.8 Thread Local Storage Pointers & Deletion Callbacks

Thread Local Storage Pointers (TLSP) are pointers stored directly in the TCB. TLSP allow each task to have its own unique set of pointers to data structures. However task deletion behavior in vanilla FreeRTOS does not automatically free the memory pointed to by TLSP. Therefore if the memory pointed to by TLSP is not explicitly freed by the user before task deletion, memory leak will occur.

ESP-IDF FreeRTOS provides the added feature of Deletion Callbacks. Deletion Callbacks are called automatically during task deletion to free memory pointed to by TLSP. Each TLSP can have its own Deletion Callback. Note that due to the `Task Deletion` behavior, there can be instances where Deletion Callbacks are called in the context of the Idle Tasks. Therefore Deletion Callbacks **should never attempt to block** and critical sections should be kept as short as possible to minimize priority inversion.

Deletion callbacks are of type `void (*TlsDeleteCallbackFunction_t)(int, void *)` where the first parameter is the index number of the associated TLSP, and the second parameter is the TLSP itself.

Deletion callbacks are set alongside TLSP by calling `vTaskSetThreadLocalStoragePointerAndDelCallback()`. Calling the vanilla FreeRTOS function `vTaskSetThreadLocalStoragePointer()` will simply set the TLSP's associated Deletion Callback to `NULL` meaning that no callback will be called for that TLSP during task deletion. If a deletion callback is `NULL`, users should manually free the memory pointed to by the associated TLSP before task deletion in order to avoid memory leak.

`FREERTOS_THREAD_LOCAL_STORAGE_POINTERS` in `menuconfig` can be used to configure the number TLSP and Deletion Callbacks a TCB will have.

For more details see *FreeRTOS API reference*.

## 4.6.9 Configuring ESP-IDF FreeRTOS

The ESP-IDF FreeRTOS can be configured using `make menuconfig` under `Component_Config/FreeRTOS`. The following section highlights some of the ESP-IDF FreeRTOS configuration options. For a full list of ESP-IDF FreeRTOS configurations, see *FreeRTOS*

`FREERTOS_UNICORE` will run ESP-IDF FreeRTOS only on **PRO\_CPU**. Note that this is **not equivalent to running vanilla FreeRTOS**. Behaviors of multiple components in ESP-IDF will be modified such as `esp32/cpu_start.c`. For more details regarding the effects of running ESP-IDF FreeRTOS on a single core, search for occurrences of `CONFIG_FREERTOS_UNICORE` in the ESP-IDF components.

`FREERTOS_THREAD_LOCAL_STORAGE_POINTERS` will define the number of Thread Local Storage Pointers each task will have in ESP-IDF FreeRTOS.

`SUPPORT_STATIC_ALLOCATION` will enable the backported functionality of `xTaskCreateStaticPinnedToCore()` in ESP-IDF FreeRTOS

`FREERTOS_ASSERT_ON_UNTESTED_FUNCTION` will trigger a halt in particular functions in ESP-IDF FreeRTOS which have not been fully tested in an SMP context.

## 4.7 High-Level Interrupts

The Xtensa architecture has support for 32 interrupts, divided over 8 levels, plus an assortment of exceptions. On the ESP32, the interrupt mux allows most interrupt sources to be routed to these interrupts using the *interrupt allocator*. Normally, interrupts will be written in C, but ESP-IDF allows high-level interrupts to be written in assembly as well, allowing for very low interrupt latencies.

### 4.7.1 Interrupt Levels

Level	Symbol	Remark
1	N/A	Exception and level 0 interrupts. Handled by ESP-IDF
2-3	N/A	Medium level interrupts. Handled by ESP-IDF
4	<code>xt_highint4</code>	Normally used by ESP-IDF debug logic
5	<code>xt_highint5</code>	Free to use
NMI	<code>xt_nmi</code>	Free to use
dbg	<code>xt_debugexception</code>	Debug exception. Called on e.g. a BREAK instruction.

Using these symbols is done by creating an assembly file (suffix `.S`) and defining the named symbols, like this:

```
.section .iram1, "ax"
.global xt_highint5
.type xt_highint5, @function
.align 4
xt_highint5:
... your code here
rsr a0, EXCSAVE_5
rfi 5
```

For a real-life example, see the `components/esp32/panic_highint_hdl.S` file; the panic handler `iuninterrupt` is implemented there.

## 4.7.2 Notes

- Do not call C code from a high-level interrupt; because these interrupts still run in critical sections, this can cause crashes. (The panic handler interrupt does call normal C code, but this is OK because there is no intention of returning to the normal code flow afterwards.)
- Make sure your assembly code gets linked in. If the interrupt handler symbol is the only symbol the rest of the code uses from this file, the linker will take the default ISR instead and not link the assembly file into the final project. To get around this, in the assembly file, define a symbol, like this:

```
.global ld_include_my_isr_file
ld_include_my_isr_file:
```

(The symbol is called `ld_include_my_isr_file` here but can have any arbitrary name not defined anywhere else.) Then, in the `component.mk`, add this file as an unresolved symbol to the `ld` command line arguments:

```
COMPONENT_ADD_LDFLAGS := -u ld_include_my_isr_file
```

This should cause the linker to always include a file defining `ld_include_my_isr_file`, causing the ISR to always be linked in.

- High-level interrupts can be routed and handled using `esp_intr_alloc` and associated functions. The handler and handler arguments to `esp_intr_alloc` must be `NULL`, however.
- In theory, medium priority interrupts could also be handled in this way. For now, ESP-IDF does not support this.

## 4.8 JTAG Debugging

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB. The document is structured as follows:

**Introduction** Introduction to the purpose of this guide.

**How it Works?** Description how ESP32, JTAG interface, OpenOCD and GDB are interconnected and working together to enable debugging of ESP32.

**Selecting JTAG Adapter** What are the criteria and options to select JTAG adapter hardware.

**Setup of OpenOCD** Procedure to install OpenOCD using prebuilt software packages for *Windows*, *Linux* and *MacOS* operating systems.

**Configuring ESP32 Target** Configuration of OpenOCD software and set up JTAG adapter hardware that will make together a debugging target.

**Launching Debugger** Steps to start up a debug session with GDB from *Eclipse* and from *Command Line*.

**Debugging Examples** If you are not familiar with GDB, check this section for debugging examples provided from *Eclipse* as well as from *Command Line*.

**Building OpenOCD from Sources** Procedure to build OpenOCD from sources for *Windows*, *Linux* and *MacOS* operating systems.

**Tips and Quirks** This section provides collection of tips and quirks related JTAG debugging of ESP32 with OpenOCD and GDB.

## 4.8.1 Introduction

The ESP32 has two powerful Xtensa cores, allowing for a great deal of variety of program architectures. The FreeRTOS OS that comes with ESP-IDF is capable of multi-core preemptive multithreading, allowing for an intuitive way of writing software.

The downside of the ease of programming is that debugging without the right tools is harder: figuring out a bug that is caused by two threads, running even simultaneously on two different CPU cores, can take a long time when all you have are printf statements. A better and in many cases quicker way to debug such problems is by using a debugger, connected to the processors over a debug port.

Espressif has ported OpenOCD to support the ESP32 processor and the multicore FreeRTOS, which will be the foundation of most ESP32 apps, and has written some tools to help with features OpenOCD does not support natively.

This document provides a guide to installing OpenOCD for ESP32 and debugging using GDB under Linux, Windows and MacOS. Except for OS specific installation procedures, the s/w user interface and use procedures are the same across all supported operating systems.

**Note:** Screenshots presented in this document have been made for Eclipse Neon 3 running on Ubuntu 16.04 LTE. There may be some small differences in what a particular user interface looks like, depending on whether you are using Windows, MacOS or Linux and / or a different release of Eclipse.

## 4.8.2 How it Works?

The key software and hardware to perform debugging of ESP32 with OpenOCD over JTAG (Joint Test Action Group) interface is presented below and includes **xtensa-esp32-elf-gdb debugger**, **OpenOCD on chip debugger** and **JTAG adapter** connected to **ESP32** target.

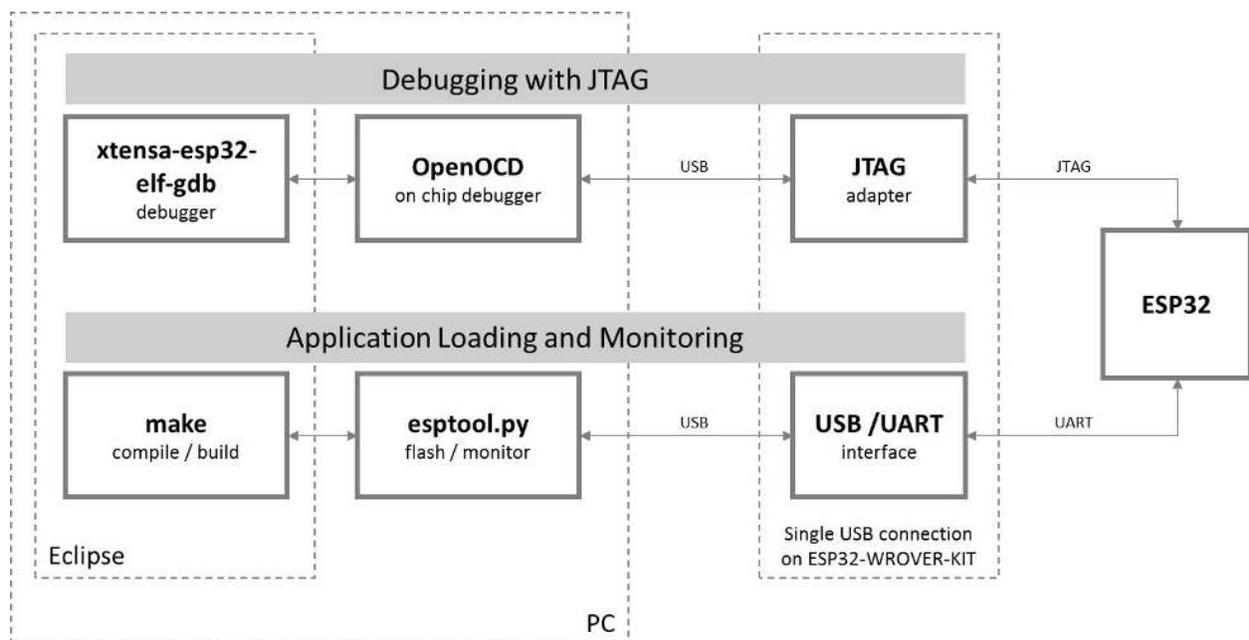


Fig. 4: JTAG debugging - overview diagram

Under “Application Loading and Monitoring” there is another software and hardware to compile, build and flash application to ESP32, as well as to provide means to monitor diagnostic messages from ESP32.

Debugging using JTAG and application loading / monitoring is integrated under the [Eclipse](#) environment, to provide quick and easy transition from writing, compiling and loading the code to debugging, back to writing the code, and so on. All the software is available for Windows, Linux and MacOS platforms.

If the [ESP32 WROVER KIT](#) is used, then connection from PC to ESP32 is done effectively with a single USB cable thanks to FT2232H chip installed on WROVER, which provides two USB channels, one for JTAG and the second for UART connection.

Depending on user preferences, both *debugger* and *make* can be operated directly from terminal / command line, instead from Eclipse.

### 4.8.3 Selecting JTAG Adapter

The quickest and most convenient way to start with JTAG debugging is by using [ESP32 WROVER KIT](#). Each version of this development board has JTAG interface already build in. No need for an external JTAG adapter and extra wiring / cable to connect JTAG to ESP32. WROVER KIT is using FT2232H JTAG interface operating at 20 MHz clock speed, which is difficult to achieve with an external adapter.

If you decide to use separate JTAG adapter, look for one that is compatible with both the voltage levels on the ESP32 as well as with the OpenOCD software. The JTAG port on the ESP32 is an industry-standard JTAG port which lacks (and does not need) the TRST pin. The JTAG I/O pins all are powered from the VDD\_3P3\_RTC pin (which normally would be powered by a 3.3V rail) so the JTAG adapter needs to be able to work with JTAG pins in that voltage range.

On the software side, OpenOCD supports a fair amount of JTAG adapters. See <http://openocd.org/doc/html/Debug-Adapter-Hardware.html> for an (unfortunately slightly incomplete) list of the adapters OpenOCD works with. This page lists SWD-compatible adapters as well; take note that the ESP32 does not support SWD. JTAG adapters that are hardcoded to a specific product line, e.g. STM32 debugging adapters, will not work.

The minimal signalling to get a working JTAG connection are TDI, TDO, TCK, TMS and GND. Some JTAG debuggers also need a connection from the ESP32 power line to a line called e.g. Vtar to set the working voltage. SRST can optionally be connected to the CH\_PD of the ESP32, although for now, support in OpenOCD for that line is pretty minimal.

### 4.8.4 Setup of OpenOCD

This step covers installation of OpenOCD binaries. If you like to build OpenOCD from sources then refer to section [Building OpenOCD from Sources](#). All OpenOCD files will be placed in `~/esp/openocd-esp32` directory. You may choose any other directory, but need to adjust respective paths used in examples.

#### Setup OpenOCD for Windows

##### Setup OpenOCD

OpenOCD for Windows / MSYS2 is available for download from Espressif website:

<https://dl.espressif.com/dl/openocd-esp32-win32-a859564.zip>

Download this file and extract `openocd-esp32` folder inside to `~/esp/` directory.

#### Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

## Related Documents

### Building OpenOCD from Sources for Windows

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

### Download Sources of OpenOCD

The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone -recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

### Install Dependencies

Install packages that are required to compile OpenOCD:

---

**Note:** Install the following packages one by one, check if installation was successful and then proceed to the next package. Resolve reported problems before moving to the next step.

---

```
pacman -S libtool
pacman -S autoconf
pacman -S automake
pacman -S texinfo
pacman -S mingw-w64-i686-libusb-compat-git
pacman -S pkg-config
```

---

**Note:** Installation of `pkg-config` is breaking operation of `esp-idf` toolchain. After building of OpenOCD it should be uninstalled. It be covered at the end of this instruction. To build OpenOCD again, you will need to run `pacman -S pkg-config` once more. This issue does not concern other packages installed in this step (before `pkg-config`).

---

### Build OpenOCD

Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

---

**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.

Once make process is successfully completed, the executable of OpenOCD will be saved in ~/esp/openocd-esp32/src/openocd directory.

Remove pkg-config, as discussed during installation of dependencies:

```
pacman -Rs pkg-config
```

## Next Steps

To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

## Setup OpenOCD for Linux

### Setup OpenOCD

OpenOCD for 64-bit Linux is available for download from Espressif website:

<https://dl.espressif.com/dl/openocd-esp32-linux64-a859564.tar.gz>

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

```
cd ~/esp
tar -xzf ~/Downloads/openocd-esp32-linux64-a859564.tar.gz
```

## Next Steps

To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

## Related Documents

### Building OpenOCD from Sources for Linux

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

## Download Sources of OpenOCD

The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone -recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

## Install Dependencies

Install packages that are required to compile OpenOCD.

---

**Note:** Install the following packages one by one, check if installation was successful and then proceed to the next package. Resolve reported problems before moving to the next step.

---

```
sudo apt-get install make
sudo apt-get install libtool
sudo apt-get install pkg-config
sudo apt-get install autoconf
sudo apt-get install automake
sudo apt-get install texinfo
sudo apt-get install libusb-1.0
```

---

**Note:**

- Version of `pkg-config` should be 0.2.3 or above.
  - Version of `autoconf` should be 2.6.4 or above.
  - Version of `automake` should be 1.9 or above.
  - When using USB-Blaster, ASIX Presto, OpenJTAG and FT2232 as adapters, drivers `libFTDI` and `FTD2XX` need to be downloaded and installed.
  - When using CMSIS-DAP, `HIDAPI` is needed.
- 

## Build OpenOCD

Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

---

**Note:**

- Should an error occur, resolve it and try again until the command `make` works.
  - If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
  - If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
  - If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
  - For details concerning compiling OpenOCD, please refer to `openocd-esp32/README`.
- 

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/openocd-esp32/bin` directory.

## Next Steps

To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

## Setup OpenOCD for MacOS

### Setup OpenOCD

OpenOCD for MacOS is available for download from Espressif website:

<https://dl.espressif.com/dl/openocd-esp32-macos-a859564.tar.gz>

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

```
cd ~/esp
tar -xzf ~/Downloads/openocd-esp32-macos-a859564.tar.gz
```

## Next Steps

To carry on with debugging environment setup, proceed to section *Configuring ESP32 Target*.

## Related Documents

### Building OpenOCD from Sources for MacOS

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

### Download Sources of OpenOCD

The sources for the ESP32-enabled variant of OpenOCD are available from Espressif GitHub under <https://github.com/espressif/openocd-esp32>. To download the sources, use the following commands:

```
cd ~/esp
git clone -recursive https://github.com/espressif/openocd-esp32.git
```

The clone of sources should be now saved in `~/esp/openocd-esp32` directory.

## Install Dependencies

Install packages that are required to compile OpenOCD using Homebrew:

```
brew install automake libtool libusb wget gcc@4.9
```

## Build OpenOCD

Proceed with configuring and building OpenOCD:

```
cd ~/esp/openocd-esp32
./bootstrap
./configure
make
```

Optionally you can add `sudo make install` step at the end. Skip it, if you have an existing OpenOCD (from e.g. another development platform), as it may get overwritten.

---

### Note:

- Should an error occur, resolve it and try again until the command `make` works.
- If there is a submodule problem from OpenOCD, please `cd` to the `openocd-esp32` directory and input `git submodule update --init`.
- If the `./configure` is successfully run, information of enabled JTAG will be printed under OpenOCD configuration summary.
- If the information of your device is not shown in the log, use `./configure` to enable it as described in `../openocd-esp32/doc/INSTALL.txt`.
- For details concerning compiling OpenOCD, please refer to `openocd-esp32/README.OSX`.

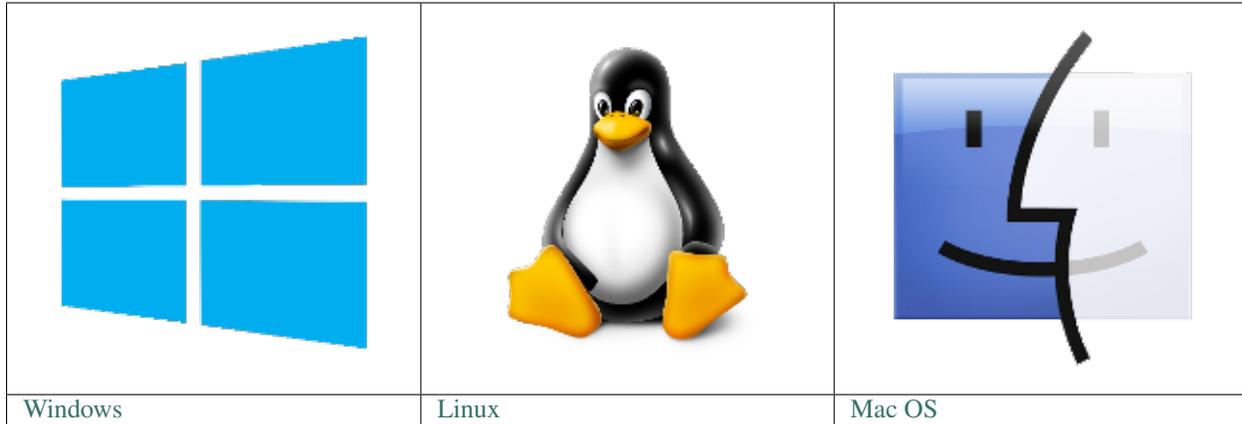
---

Once `make` process is successfully completed, the executable of OpenOCD will be saved in `~/esp/openocd-esp32/src/openocd` directory.

## Next Steps

To carry on with debugging environment setup, proceed to section [Configuring ESP32 Target](#).

Pick up your OS below and follow provided instructions to setup OpenOCD.



After installation is complete, get familiar with two key directories inside `openocd-esp32` installation folder:

- `bin` containing OpenOCD executable
- `share\openocd\scripts` containing configuration files invoked together with OpenOCD as command line parameters

**Note:** Directory names and structure above are specific to binary distribution of OpenOCD. They are used in examples of invoking OpenOCD throughout this guide. Directories for OpenOCD build from sources are different, so the way to invoke OpenOCD. For details see [Building OpenOCD from Sources](#).

## 4.8.5 Configuring ESP32 Target

Once OpenOCD is installed, move to configuring ESP32 target (i.e ESP32 board with JTAG interface). You will do it in the following three steps:

- Configure and connect JTAG interface
- Run OpenOCD
- Upload application for debugging

### Configure and connect JTAG interface

This step depends on JTAG and ESP32 board you are using - see the two cases described below.

#### Configure WROVER JTAG Interface

All versions of ESP32 WROVER KIT boards have JTAG functionality build in. Putting it to work requires setting jumpers to enable JTAG functionality, setting SPI flash voltage and configuring USB drivers. Please refer to step by step instructions below.

#### Configure Hardware

1. Enable on-board JTAG functionality by setting JP8 according to [ESP-WROVER-KIT V3 Getting Started Guide](#), section *Setup Options*.

2. Verify if ESP32 pins used for JTAG communication are not connected to some other h/w that may disturb JTAG operation:

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

## Configure USB Drivers

Install and configure USB drivers, so OpenOCD is able to communicate with JTAG interface on ESP32 WROVER KIT board as well as with UART interface used to upload application for flash. Follow steps below specific to your operating system.

---

**Note:** ESP32 WROVER KIT uses an FT2232 adapter. The following instructions can also be used for other FT2232 based JTAG adapters.

---

## Windows

1. Using standard USB A / micro USB B cable connect ESP32 WROVER KIT to the computer. Switch the WROVER KIT on.
2. Wait until USB ports of WROVER KIT are recognized by Windows and drives are installed. If they do not install automatically, then then download them from <http://www.ftdichip.com/Drivers/D2XX.htm> and install manually.
3. Download Zadig tool (Zadig\_X.X.exe) from <http://zadig.akeo.ie/> and run it.
4. In Zadig tool go to “Options” and check “List All Devices”.
5. Check the list of devices that should contain two WROVER specific USB entries: “Dual RS232-HS (Interface 0)” and “Dual RS232-HS (Interface 1)”. The driver name would be “FTDIBUS (vxxxx)” and USB ID: 0403 6010.
6. The first device (Dual RS232-HS (Interface 0)) is connected to the JTAG port of the ESP32. Original “FTDIBUS (vxxxx)” driver of this device should be replaced with “WinUSB (v6xxxx)”. To do so, select “Dual RS232-HS (Interface 0) and reinstall attached driver to the “WinUSB (v6xxxx)”, see picture above.

---

**Note:** Do not change the second device “Dual RS232-HS (Interface 1)”. It is routed to ESP32’s serial port (UART) used for upload of application to ESP32’s flash.

---

Now ESP32 WROVER KIT’s JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

## Linux

1. Using standard USB A / micro USB B cable connect ESP32 WROVER KIT board to the computer. Power on the board.

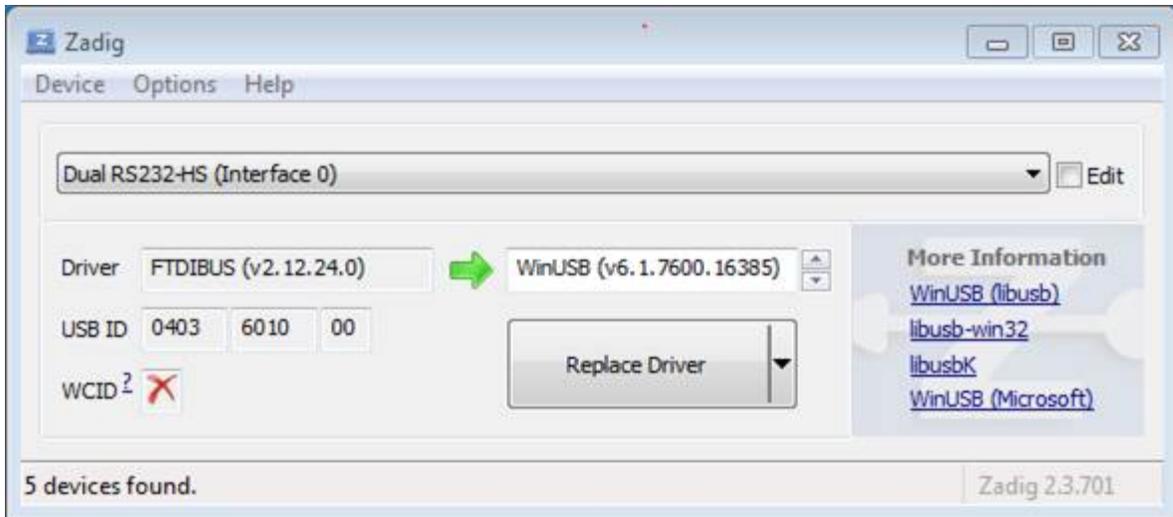


Fig. 5: Configuration of JTAG USB driver in Zadig tool

2. Open a terminal, enter `ls -l /dev/ttyUSB*` command and check, if board's USB ports are recognized by the OS. You are looking for similar result:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Jul 10 19:04 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 Jul 10 19:04 /dev/ttyUSB1
```

3. Following section “Permissions delegation” in OpenOCD’s README, set up the access permissions to both USB ports.
4. Log off and login, then cycle the power to the board to make the changes effective. In terminal enter again `ls -l /dev/ttyUSB*` command to verify, if group-owner has changed from dialout to plugdev:

```
user-name@computer-name:~/esp$ ls -l /dev/ttyUSB*
crw-rw-r-- 1 root plugdev 188, 0 Jul 10 19:07 /dev/ttyUSB0
crw-rw-r-- 1 root plugdev 188, 1 Jul 10 19:07 /dev/ttyUSB1
```

If you see similar result and you are a member of `plugdev` group, then the set up is complete.

The `/dev/ttyUSBn` interface with lower number is used for JTAG communication. The other interface is routed to ESP32’s serial port (UART) used for upload of application to ESP32’s flash.

Now ESP32 WROVER KIT’s JTAG interface should be available to the OpenOCD. To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

## MacOS

On macOS, using 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 doesn't load itself for channel B of FT2232 chip, which is the channel used for JTAG on WROVER KIT.

### Manually unloading the driver

1. Install FTDI driver from <http://www.ftdichip.com/Drivers/VCP.htm>
2. Connect USB cable to the WROVER KIT.
3. Unload the serial port driver:

```
sudo kextunload -b com.FTDI.driver.FTDIUSBSerialDriver
```

In some cases you may need to unload Apple's FTDI driver as well:

```
sudo kextunload -b com.apple.driver.AppleUSBFTDI
```

4. Run OpenOCD (paths are given for downloadable OpenOCD archive):

```
bin/openocd -s share/openocd/scripts -f interface/ftdi/esp32_devkitj_v1.cfg -f ↵  
↳board/esp-wroom-32.cfg
```

Or, if OpenOCD was built from source:

```
src/openocd -s tcl -f interface/ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.  
↳cfg
```

5. In another terminal window, load FTDI serial port driver again:

```
sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
```

Note that if you need to restart OpenOCD, there is no need to unload FTDI driver again — just stop OpenOCD and start it again. The driver only needs to be unloaded if WROVER KIT was reconnected or power was toggled.

This procedure can be wrapped into a shell script, if desired.

### Modifying FTDI driver

In a nutshell, this approach requires modification to FTDI driver configuration file, which prevents the driver from being loaded for channel B of FT2232H.

---

**Note:** Other boards may use channel A for JTAG, so use this option with caution.

---

**Warning:** This approach also needs signature verification of drivers to be disabled, so may not be acceptable for all users.

1. Open FTDI driver configuration file using a text editor (note `sudo`):

```
sudo nano /Library/Extensions/FTDIUSBSerialDriver.kext/Contents/Info.plist
```

2. Find and delete the following lines:

```

<key>FT2232H_B</key>
<dict>
  <key>CFBundleIdentifier</key>
  <string>com.FTDI.driver.FTDIUSBSerialDriver</string>
  <key>IOClass</key>
  <string>FTDIUSBSerialDriver</string>
  <key>IOProviderClass</key>
  <string>IOUSBInterface</string>
  <key>bConfigurationValue</key>
  <integer>1</integer>
  <key>bInterfaceNumber</key>
  <integer>1</integer>
  <key>bcdDevice</key>
  <integer>1792</integer>
  <key>idProduct</key>
  <integer>24592</integer>
  <key>idVendor</key>
  <integer>1027</integer>
</dict>

```

3. Save and close the file
4. Disable driver signature verification:
  1. Open Apple logo menu, choose “Restart…”
  2. When you hear the chime after reboot, press CMD+R immediately
  3. Once Recovery mode starts up, open Terminal
  4. Run the command:

```
csrutil enable --without kext
```

5. Restart again

After these steps, serial port and JTAG can be used at the same time.

To carry on with debugging environment setup, proceed to section [Run OpenOCD](#).

## Configure Other JTAG Interface

Refer to section [Selecting JTAG Adapter](#) for guidance what JTAG interface to select, so it is able to operate with OpenOCD and ESP32. Then follow three configuration steps below to get it working.

## Configure Hardware

1. Identify all pins / signals on JTAG interface and ESP32 board, that should be connected to establish communication.

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

2. Verify if ESP32 pins used for JTAG communication are not connected to some other h/w that may disturb JTAG operation.
3. Connect identified pin / signals of ESP32 and JTAG interface.

### Configure Drivers

You may need to install driver s/w to make JTAG work with computer. Refer to documentation of JTAG adapter, that should provide related details.

### Connect

Connect JTAG interface to the computer. Power on ESP32 and JTAG interface boards. Check if JTAG interface is visible by computer.

To carry on with debugging environment setup, proceed to section *Run OpenOCD*.

### Run OpenOCD

Once target is configured and connected to computer, you are ready to launch OpenOCD.

Open terminal, go to directory where OpenOCD is installed and start it up:

```
cd ~/esp/openocd-esp32
bin/openocd -s share/openocd/scripts -f interface/ftdi/esp32_devkitj_v1.cfg -f board/
↳ esp-wroom-32.cfg
```

---

**Note:** The files provided after `-f` above, are specific for ESP-WROVER-KIT with ESP-WROOM-32 module. You may need to provide different files depending on used hardware, For guidance see *Configuration of OpenOCD for specific target*.

---

You should now see similar output (this log is for ESP32 WROVER KIT):

```
user-name@computer-name:~/esp/openocd-esp32$ bin/openocd -s share/openocd/scripts -f_
↳ interface/ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.cfg
Open On-Chip Debugger 0.10.0-dev-ged7b1a9 (2017-07-10-07:16)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
none separate
adapter speed: 20000 kHz
force hard breakpoints
Info : ftdi: if you experience problems at higher adapter clocks, try the command
↳ "ftdi_tdo_sample_edge falling"
Info : clock speed 20000 kHz
Info : JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),_
↳ part: 0x2003, ver: 0x1)
Info : JTAG tap: esp32.cpu1 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica),_
↳ part: 0x2003, ver: 0x1)
Info : esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
Info : esp32: Core was reset (pwrstat=0x5F, after clear 0x0F).
```

- If there is an error indicating permission problems, please see the “Permissions delegation” bit in the OpenOCD README file in `~/esp/openocd-esp32` directory.

- In case there is an error finding configuration files, e.g. `Can't find interface/ftdi/esp32_devkitj_v1.cfg`, check the path after `-s`. This path is used by OpenOCD to look for the files specified after `-f`. Also check if the file is indeed under provided path.
- If you see JTAG errors (...all ones/...all zeroes) please check your connections, whether no other signals are connected to JTAG besides ESP32's pins, and see if everything is powered on.

## Upload application for debugging

Build and upload your application to ESP32 as usual, see *Build and Flash*.

Another option is to write application image to flash using OpenOCD via JTAG with commands like this:

```
cd ~/esp/openocd-esp32
bin/openocd -s share/openocd/scripts -f interface/ftdi/esp32_devkitj_v1.cfg -f board/
↳ esp-wroom-32.cfg -c "program_esp32 filename.bin 0x10000 verify exit"
```

OpenOCD flashing command `program_esp32` has the following format:

```
program_esp32 <image_file> <offset> [verify] [reset] [exit]
```

- `image_file` - Path to program image file.
- `offset` - Offset in flash bank to write image.
- `verify` - Optional. Verify flash contents after writing.
- `reset` - Optional. Reset target after programing.
- `exit` - Optional. Finally exit OpenOCD.

You are now ready to start application debugging. Follow steps described in section below.

## 4.8.6 Launching Debugger

The toolchain for ESP32 features GNU Debugger, in short GDB. It is available with other toolchain programs under filename `xtensa-esp32-elf-gdb`. GDB can be called and operated directly from command line in a terminal. Another option is to call it from within IDE (like Eclipse, Visual Studio Code, etc.) and operate indirectly with help of GUI instead of typing commands in a terminal.

Both options of using debugger are discussed under links below.

- *Eclipse*
- *Command Line*

It is recommended to first check if debugger works from *Command Line* and then move to using *Eclipse*.

## 4.8.7 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 though the code, call stack and threads*
2. *Setting and clearing breakpoints*
3. *Halting the target manually*
4. *Stepping through the code*

5. *Checking and setting memory*
6. *Watching and setting program variables*
7. *Setting conditional breakpoints*

Similar debugging actions are provided using GDB from *Command Line*.

Before proceeding to examples, set up your ESP32 target and load it with `get-started/blink`.

## 4.8.8 Building OpenOCD from Sources

Please refer to separate documents listed below, that describe build process.

---

**Note:** 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 the path to configuration files to `-s tcl`.

Example of invoking OpenOCD build locally from sources:

```
src/openocd -s tcl -f interface/ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.cfg
```

---

## 4.8.9 Tips and Quirks

This section provides collection of links to all tips and quirks referred to from various parts of this guide.

- *Breakpoints and watchpoints available*
- *What else should I know about breakpoints?*
- *Why stepping with “next” does not bypass subroutine calls?*
- *Support options for OpenOCD at compile time*
- *FreeRTOS support*
- *Why to set SPI flash voltage in OpenOCD configuration?*
- *Optimize JTAG speed*
- *What is the meaning of debugger’s startup commands?*
- *Configuration of OpenOCD for specific target*
- *How debugger resets ESP32?*
- *Do not use JTAG pins for something else*
- *Reporting issues with OpenOCD / GDB*

## 4.8.10 Related Documents

### Using Debugger

This section covers configuration and running debugger either from *Eclipse* or *Command Line*. It is recommended to first check if debugger works from *Command Line* and then move to using Eclipse.

## Eclipse

Debugging functionality is provided out of box in standard Eclipse installation. Another option is to use pluggins like “GDB Hardware Debugging” plugin. We have found this plugin quite convenient and decided to use throughout this guide.

To begin with, install “GDB Hardware Debugging” plugin by opening Eclipse and going to *Help > Install New Software*.

Once installation is complete, configure debugging session following steps below. Please note that some of configuration parameters are generic and some are project specific. This will be shown below by configuring debugging for “blink” example project. If not done already, add this project to Eclipse workspace following guidance in section *Build and Flash with Eclipse IDE*. The source of `get-started/blink` application is available in `examples` directory of ESP-IDF repository.

1. In Eclipse go to *Run > Debug Configuration*. A new window will open. In the window’s left pane double click “GDB Hardware Debugging” (or select “GDB Hardware Debugging” and press the “New” button) to create a new configuration.
2. In a form that will show up on the right, enter the “Name:” of this configuration, e.g. “Blink checking”.
3. On the “Main” tab below, under “Project:”, press “Browse” button and select the “blink” project.
4. In next line “C/C++ Application:” press “Browse” button and select “blink.elf” file. If “blink.elf” is not there, then likely this project has not been build yet. See *Build and Flash with Eclipse IDE* how to do it.
5. Finally, under “Build (if required) before launching” click “Disable auto build”.

A sample window with settings entered in points 1 - 5 is shown below.

6. Click “Debugger” tab. In field “GDB Command” enter `xtensa-esp32-elf-gdb` to invoke debugger.
7. Change default configuration of “Remote host” by entering 3333 under the “Port number”.

Configuration entered in points 6 and 7 is shown on the following picture.

8. The last tab to that requires changing of default configuration is “Startup”. Under “Initialization Commands” uncheck “Reset and Delay (seconds)” and “Halt”. Then, in entry field below, type `mon reset halt` and `x $a1=0` (in two separate lines).

**Note:** If you want to update image in the flash automatically before starting new debug session add the following lines of commands at the beginning of “Initialization Commands” textbox:

```
mon reset halt
mon program_esp32 ${workspace_loc:blink/build/blink.bin} 0x10000 verify
```

For description of `program_esp32` command see *Upload application for debugging*.

9. Under “Load Image and Symbols” uncheck “Load image” option.
10. Further down on the same tab, establish an initial breakpoint to halt CPUs after they are reset by debugger. The plugin will set this breakpoint at the beginning of the function entered under “Set break point at:”. Checkout this option and enter `app_main` in provided field.
11. Checkout “Resume” option. This will make the program to resume after `mon reset halt` is invoked per point 8. The program will then stop at breakpoint inserted at `app_main`.

Configuration described in points 8 - 11 is shown below.

If the “Startup” sequence looks convoluted and respective “Initialization Commands” are not clear to you, check *What is the meaning of debugger’s startup commands?* for additional explanation.

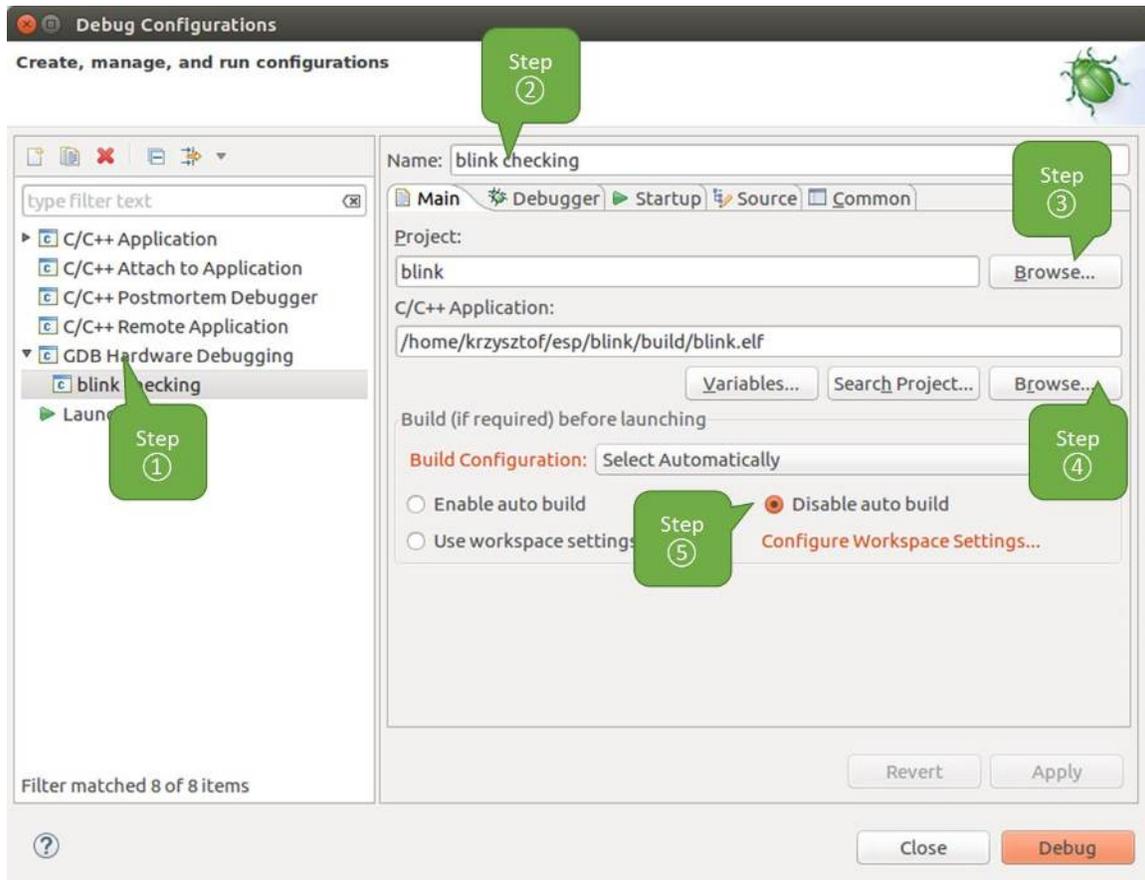


Fig. 6: Configuration of GDB Hardware Debugging - Main tab

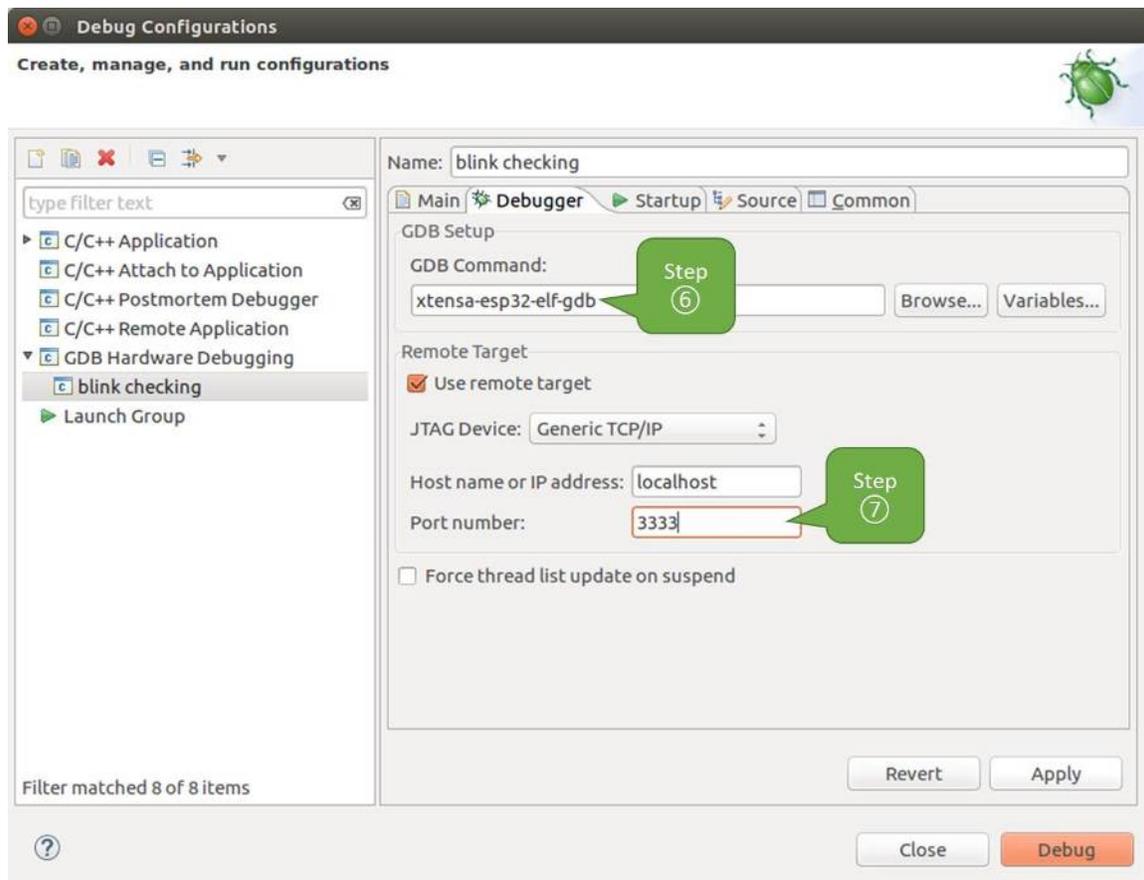


Fig. 7: Configuration of GDB Hardware Debugging - Debugger tab

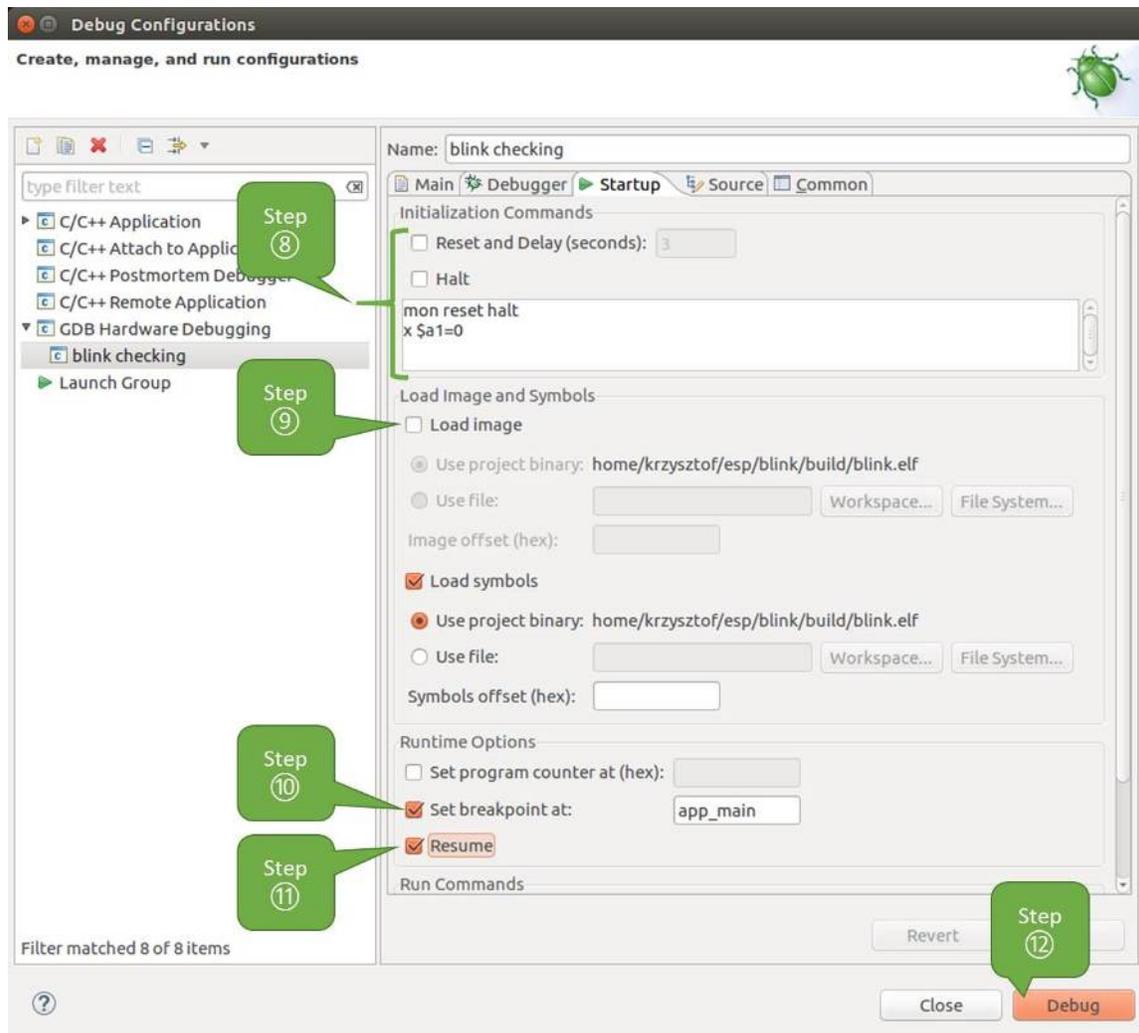


Fig. 8: Configuration of GDB Hardware Debugging - Startup tab

- If you previously completed *Configuring ESP32 Target* steps described above, so the target is running and ready to talk to debugger, go right to debugging by pressing “Debug” button. Otherwise press “Apply” to save changes, go back to *Configuring ESP32 Target* and return here to start debugging.

Once all 1 - 12 configuration steps are satisfied, the new Eclipse perspective called “Debug” will open as shown on example picture below.

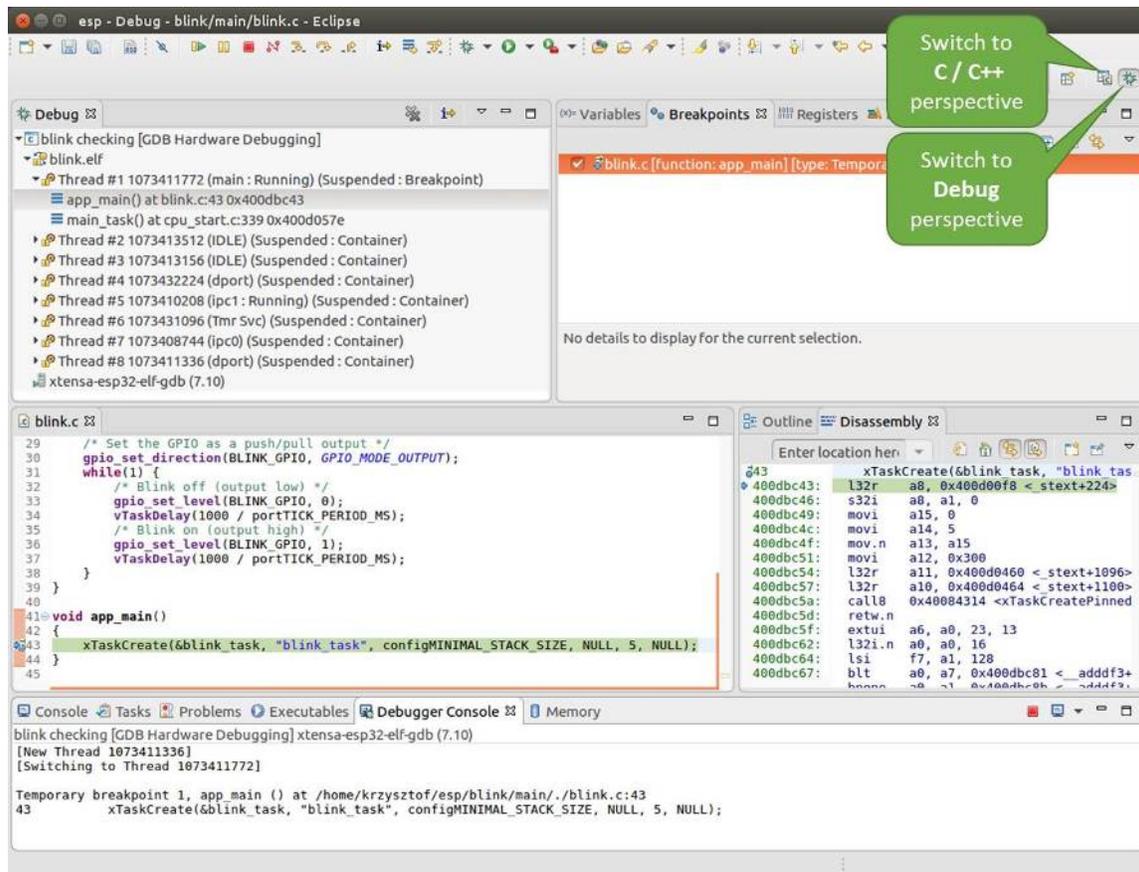


Fig. 9: Debug Perspective in Eclipse

If you are not quite sure how to use GDB, check *Eclipse* example debugging session in section *Debugging Examples*.

## Command Line

- To be able start debugging session, the target should be up and running. If not done already, complete steps described under *Configuring ESP32 Target*.
- Open a new terminal session and go to directory that contains project for debugging, e.g.

```
cd ~/esp/blink
```

- When launching a debugger, you will need to provide couple of configuration parameters and commands. Instead of entering them one by one in command line, create a configuration file and name it `gdbinit`:

```
target remote :3333
mon reset halt
```

(continues on next page)

(continued from previous page)

```
thb app_main
x $al=0
c
```

Save this file in current directory.

For more details what's inside `gdbinit` file, see [What is the meaning of debugger's startup commands?](#)

4. Now you are ready to launch GDB. Type the following in terminal:

```
xtensa-esp32-elf-gdb -x gdbinit build/blink.elf
```

5. If previous steps have been done correctly, you will see a similar log concluded with (gdb) prompt:

```
user-name@computer-name:~/esp/blink$ xtensa-esp32-elf-gdb -x gdbinit build/blink.
↳elf
GNU gdb (crosstool-NG crosstool-ng-1.22.0-61-gab8375a) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_pc-linux-gnu --target=xtensa-
↳esp32-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build/blink.elf...done.
0x400d10d8 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/
↳components/esp32/./freertos_hooks.c:52
52      asm("waiti 0");
JTAG tap: esp32.cpu0 tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica), part:␣
↳0x2003, ver: 0x1)
JTAG tap: esp32.slave tap/device found: 0x120034e5 (mfg: 0x272 (Tensilica), part:␣
↳0x2003, ver: 0x1)
esp32: Debug controller was reset (pwrstat=0x5F, after clear 0x0F).
esp32: Core reset (pwrstat=0x5F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x5000004B (active) APP_CPU: PC=0x00000000
esp32: target state: halted
esp32: Core was reset (pwrstat=0x1F, after clear 0x0F).
Target halted. PRO_CPU: PC=0x40000400 (active) APP_CPU: PC=0x40000400
esp32: target state: halted
Hardware assisted breakpoint 1 at 0x400db717: file /home/user-name/esp/blink/main/
↳./blink.c, line 43.
0x0: 0x00000000
Target halted. PRO_CPU: PC=0x400DB717 (active) APP_CPU: PC=0x400D10D8
[New Thread 1073428656]
[New Thread 1073413708]
[New Thread 1073431316]
[New Thread 1073410672]
[New Thread 1073408876]
[New Thread 1073432196]
[New Thread 1073411552]
```

(continues on next page)

(continued from previous page)

```
[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 line from bottom that shows debugger halting at breakpoint established in `gdbinit` file at function `app_main()`. Since the processor is halted, the LED should not be blinking. If this is what you see as well, you are ready to start debugging.

If you are not quite sure how to use GDB, check *Command Line* example debugging session in section *Debugging Examples*.

## Debugging Examples

This section describes debugging with GDB from *Eclipse* as well as from *Command Line*.

### Eclipse

Verify if your target is ready and loaded with `get-started/blink` example. Configure and start debugger following steps in section *Eclipse*. Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`.

### Examples in this section

1. *Navigating though 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 though 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.

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.

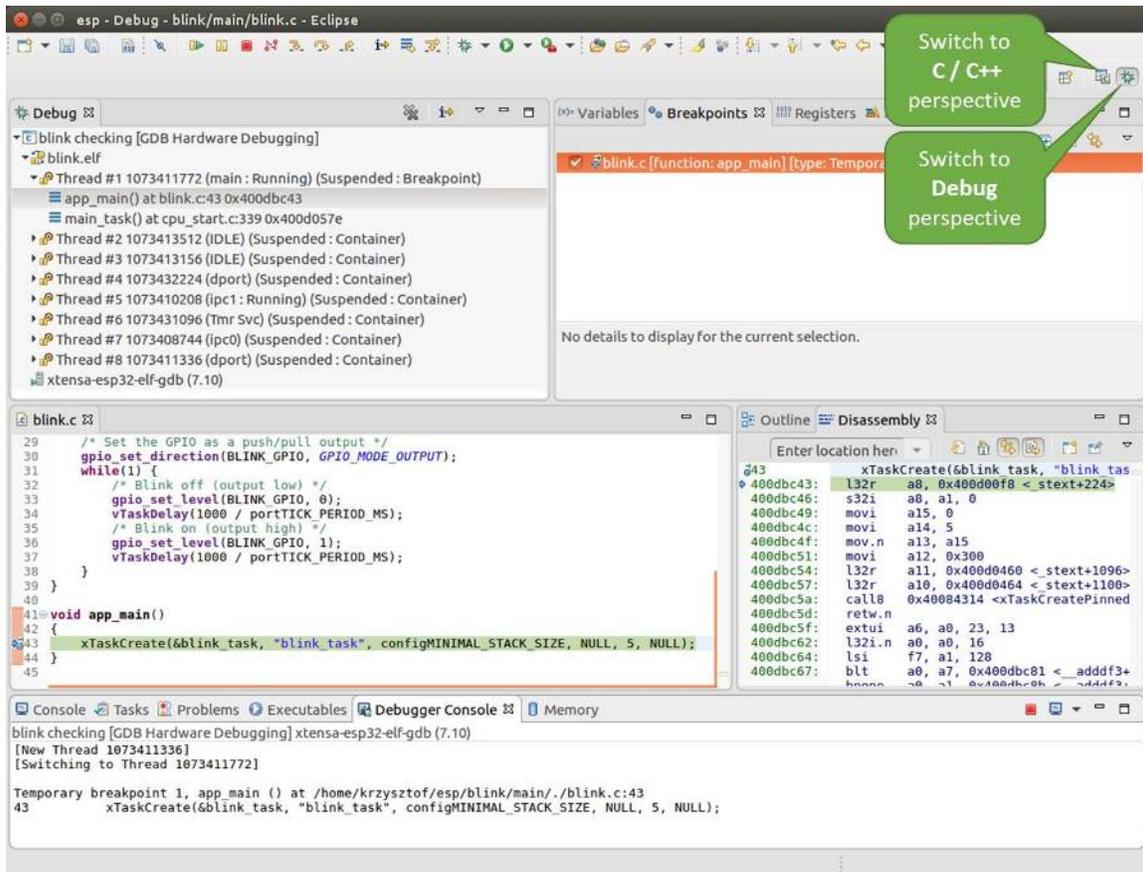


Fig. 10: Debug Perspective in Eclipse

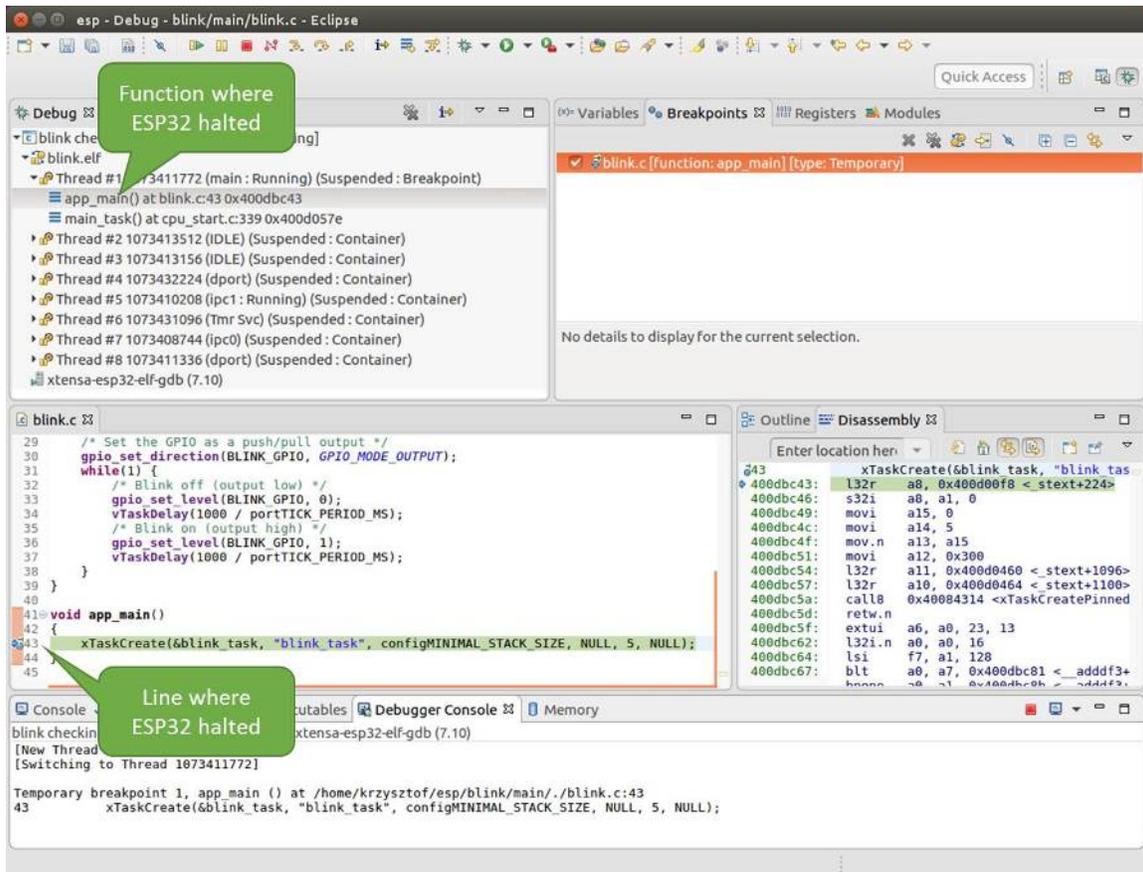


Fig. 11: Target halted during debugging

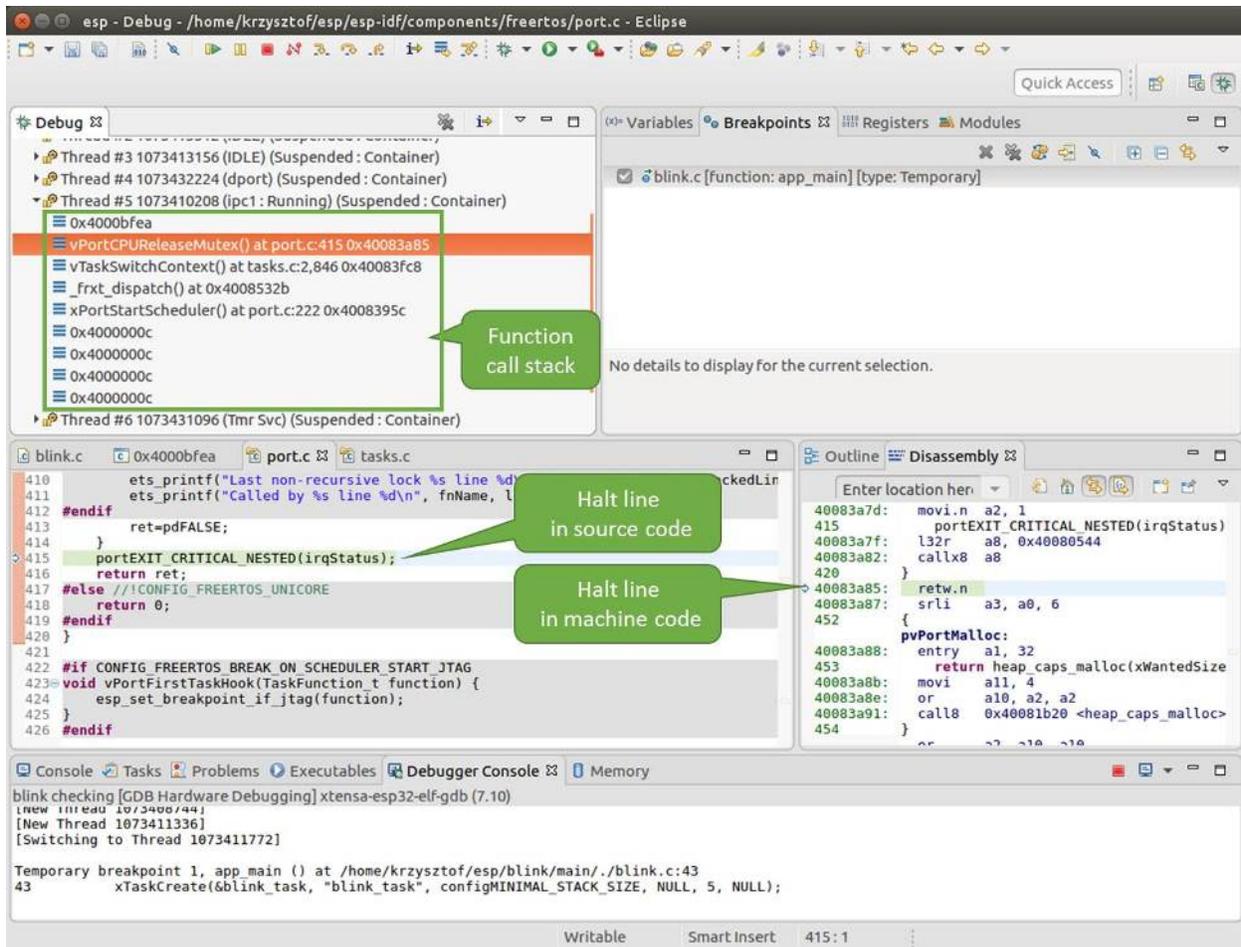


Fig. 12: Navigate through the call stack

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 / 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. To do so, hold the "Control" on the keyboard and double click on number 33 in file `blink.c` file. A dialog will open where you can confirm your selection by pressing "OK" button. If you do not like to see the dialog just double click the line number. Set another breakpoint in line 36.

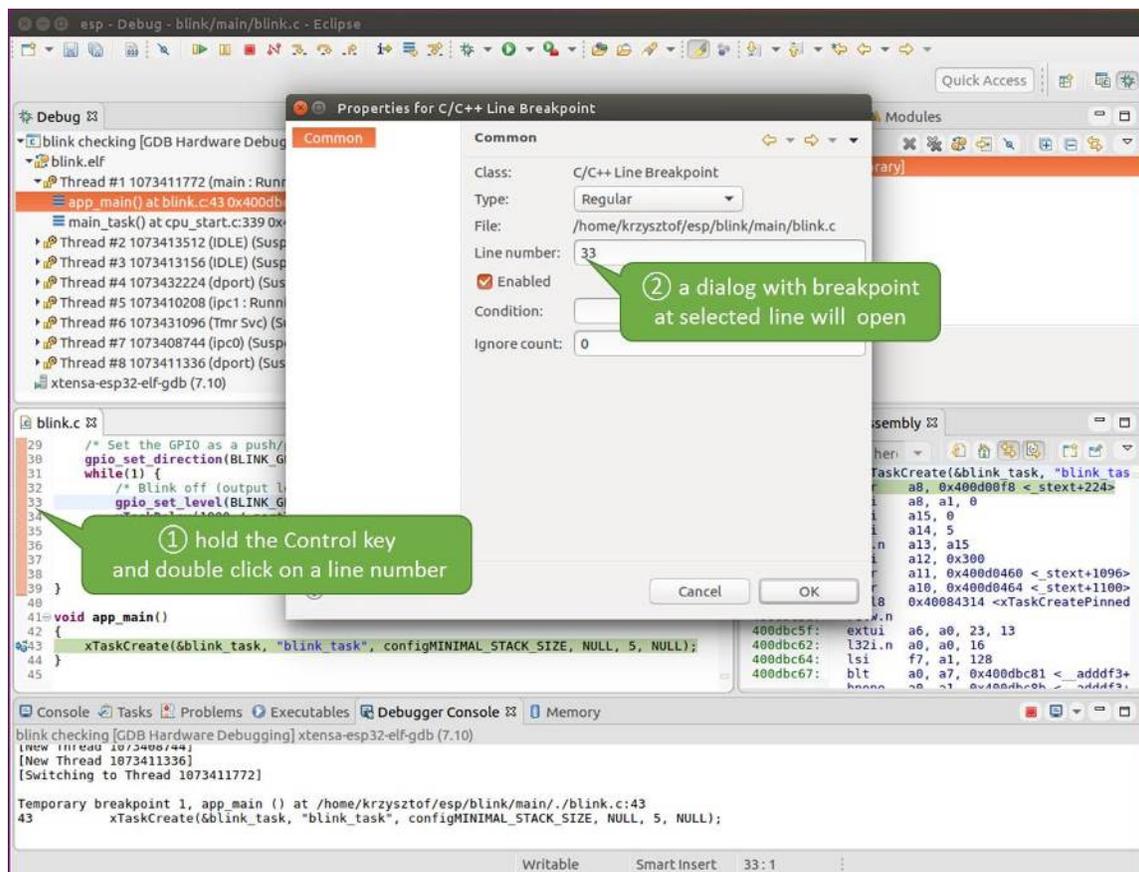


Fig. 13: Setting a breakpoint

Information how many breakpoints are set and where is shown in window "Breakpoints" on top right. Click "Show Breakpoints Supported by Selected Target" to refresh this list. Besides the two just set breakpoints the list may contain temporary breakpoint at function `app_main()` established at debugger start. As maximum two breakpoints are allowed (see *Breakpoints and watchpoints available*), you need to delete it, or debugging will fail.

If you now click "Resume" (click `blink_task()` under "Tread #8", if "Resume" button is grayed out), the processor

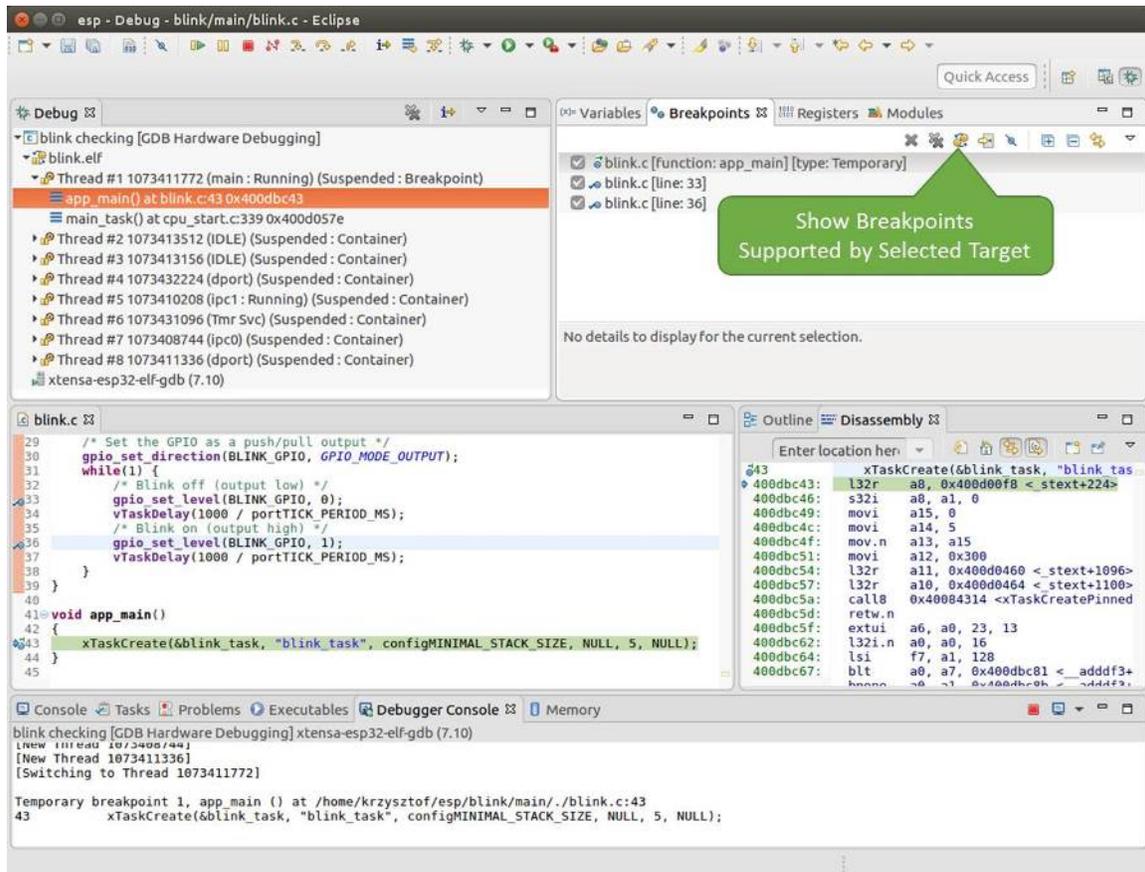


Fig. 14: Three breakpoints are set / maximum two are allowed

will run and halt at a breakpoint. Clicking “Resume” another time will make it run again, halt on second breakpoint, and so on.

You will be also able to see that LED is changing the state after each click to “Resume” program execution.

Read more about breakpoints under *Breakpoints and watchpoints available* and *What else should I know about breakpoints?*

### Halting the target manually

When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by pressing “Suspend” button.

To check it, delete all breakpoints and click “Resume”. Then click “Suspend”. Application will be halted at some random point and LED will stop blinking. Debugger will expand tread and highlight the line of code where application halted.

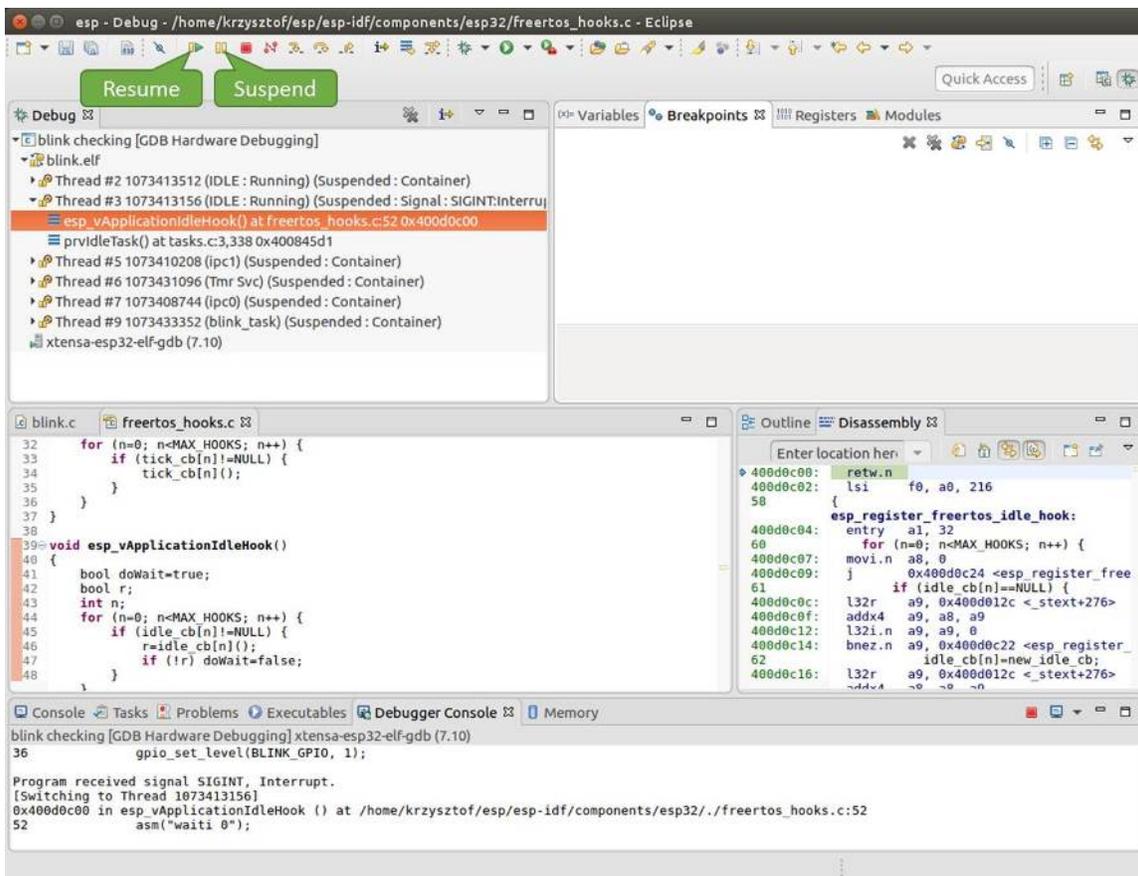


Fig. 15: Target halted manually

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.

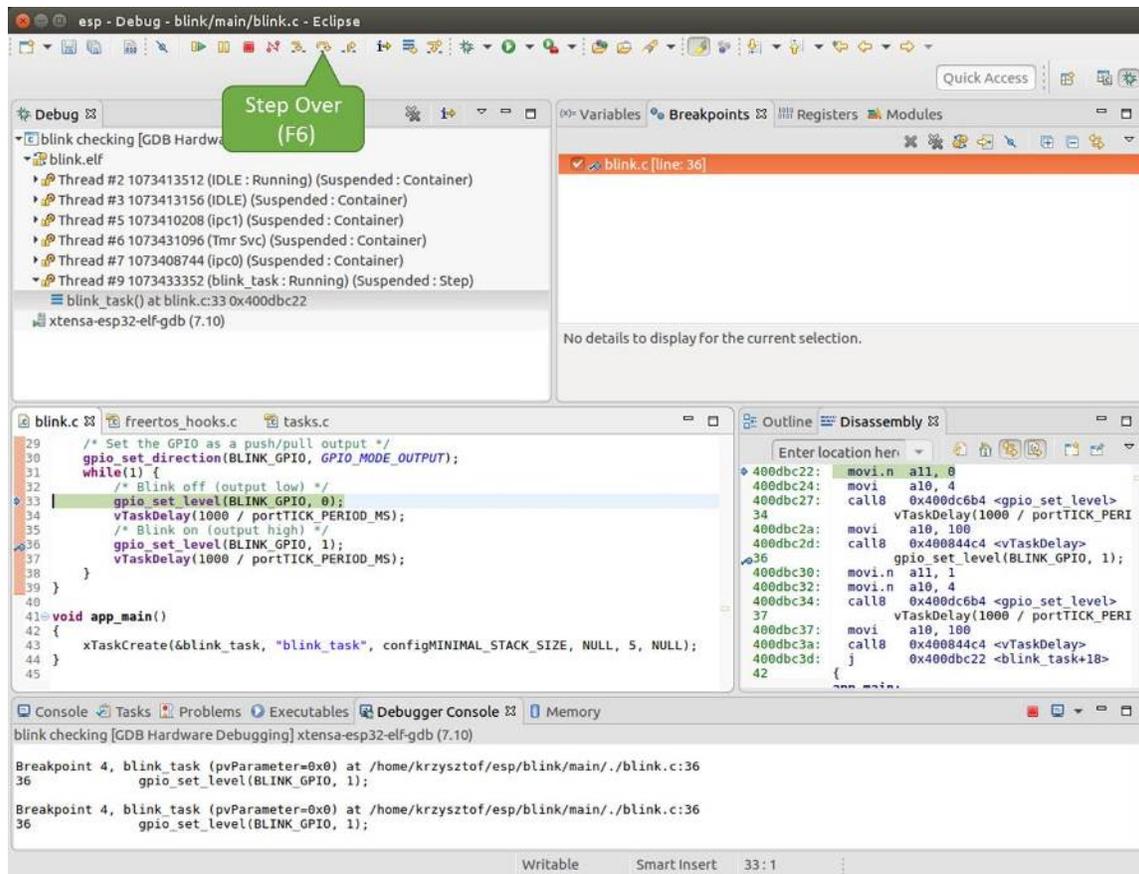


Fig. 16: Stepping through the code with “Step Over (F6)”

If you press “Step Into (F5)” instead, then debugger will step inside subroutine calls.

In this particular case debugger stepped inside `gpio_set_level(BLINK_GPIO, 0)` and effectively moved to `gpio.c` driver code.

See *Why stepping with “next” does not bypass subroutine calls?* for potential limitation of using `next` command.

## Checking and setting memory

To display or set contents of memory use “Memory” tab at the bottom of “Debug” perspective.

With the “Memory” tab, we will read from and write to the memory location `0x3FF44004` labeled as `GPIO_OUT_REG` used to set and clear individual GPIO’s. For more information please refer to [ESP32 Technical](#)

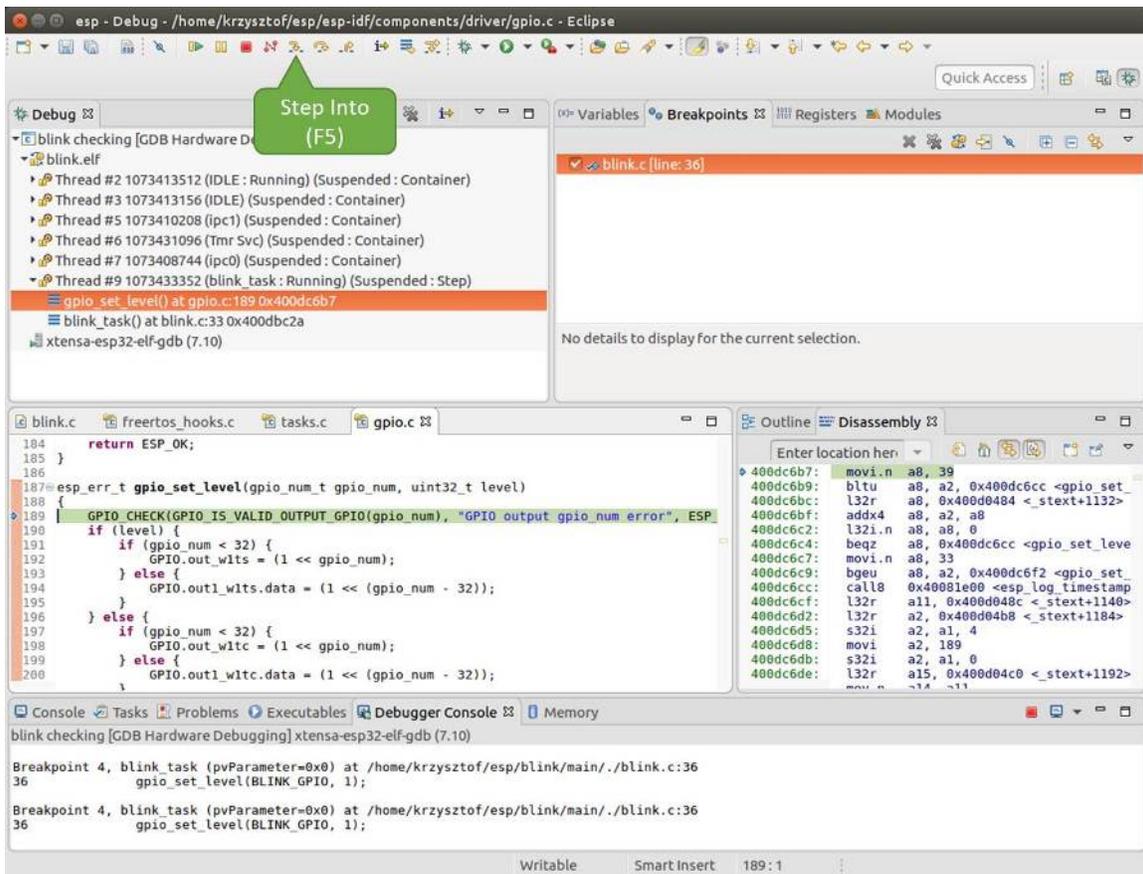


Fig. 17: Stepping through the code with “Step Into (F5)”

Reference Manual, chapter IO\_MUX and GPIO Matrix.

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Click “Memory” tab and then “Add Memory Monitor” button. Enter `0x3FF44004` in provided dialog.

Now resume program by pressing F8 and observe “Monitor” tab.

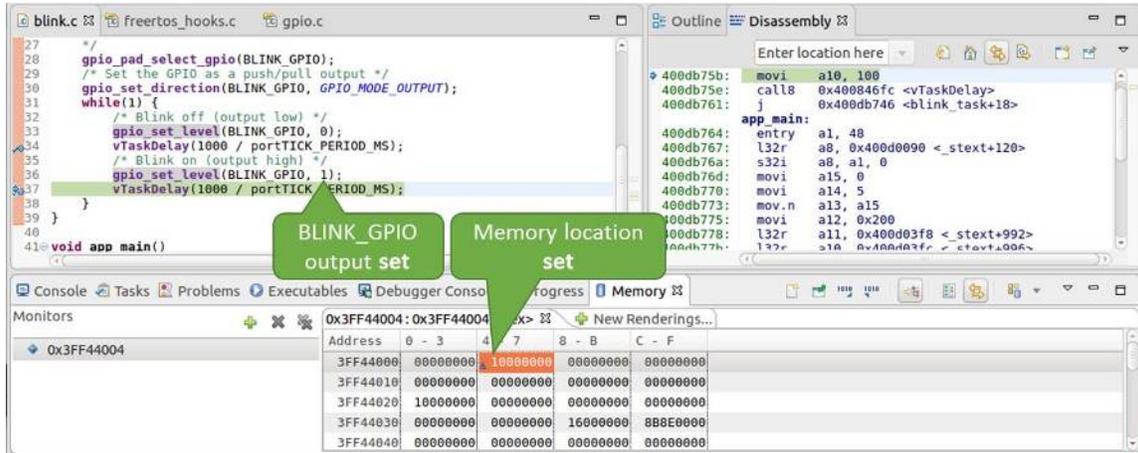


Fig. 18: Observing memory location `0x3FF44004` changing one bit to ON”

You should see one bit being flipped over at memory location `0x3FF44004` (and LED changing the state) each time F8 is pressed.

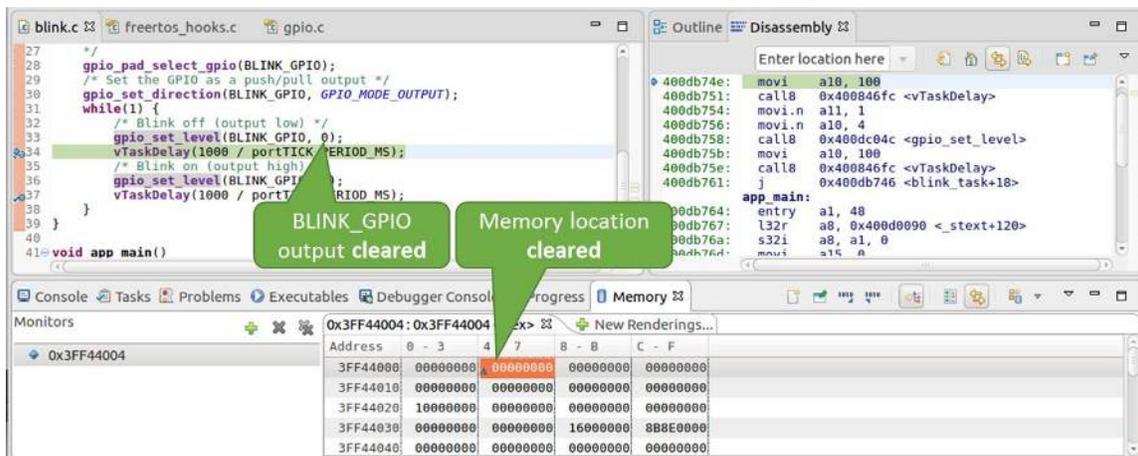


Fig. 19: Observing memory location `0x3FF44004` changing one bit to ON”

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 `loop(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.

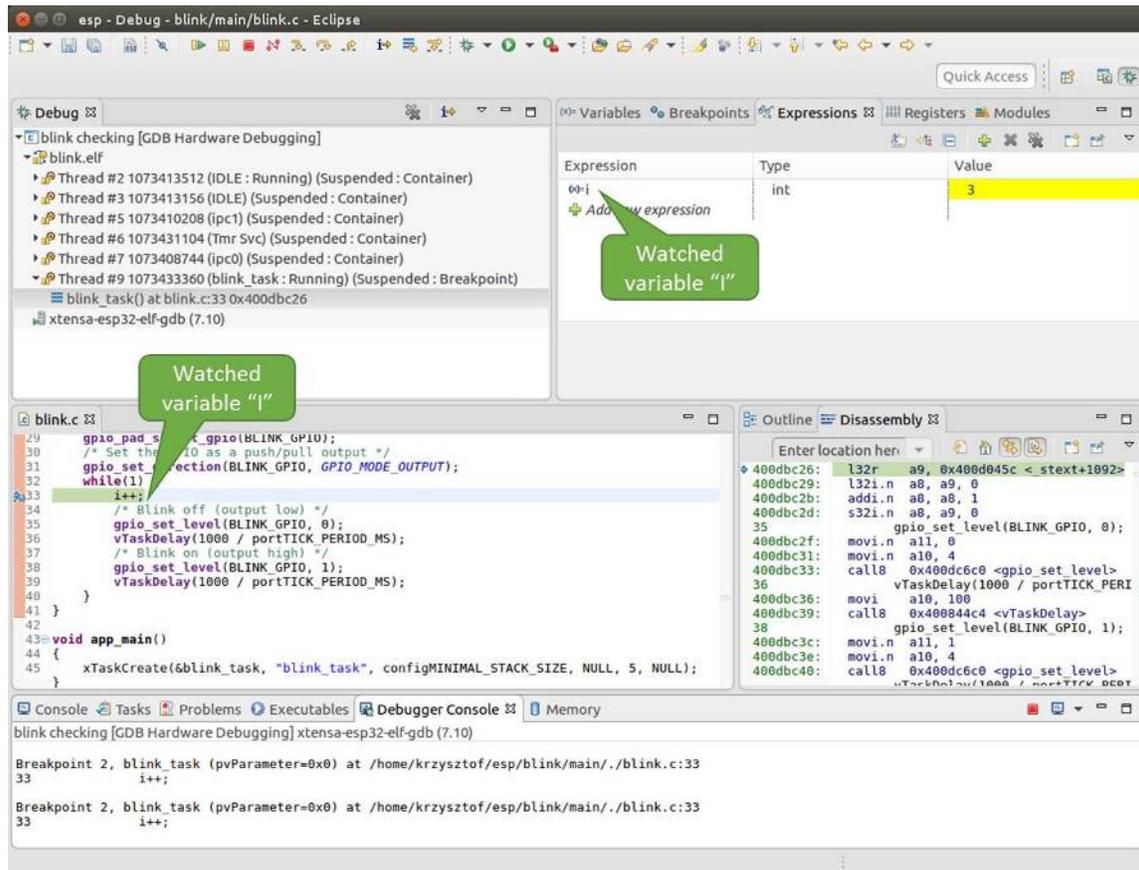


Fig. 20: Watching program variable “i”

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.

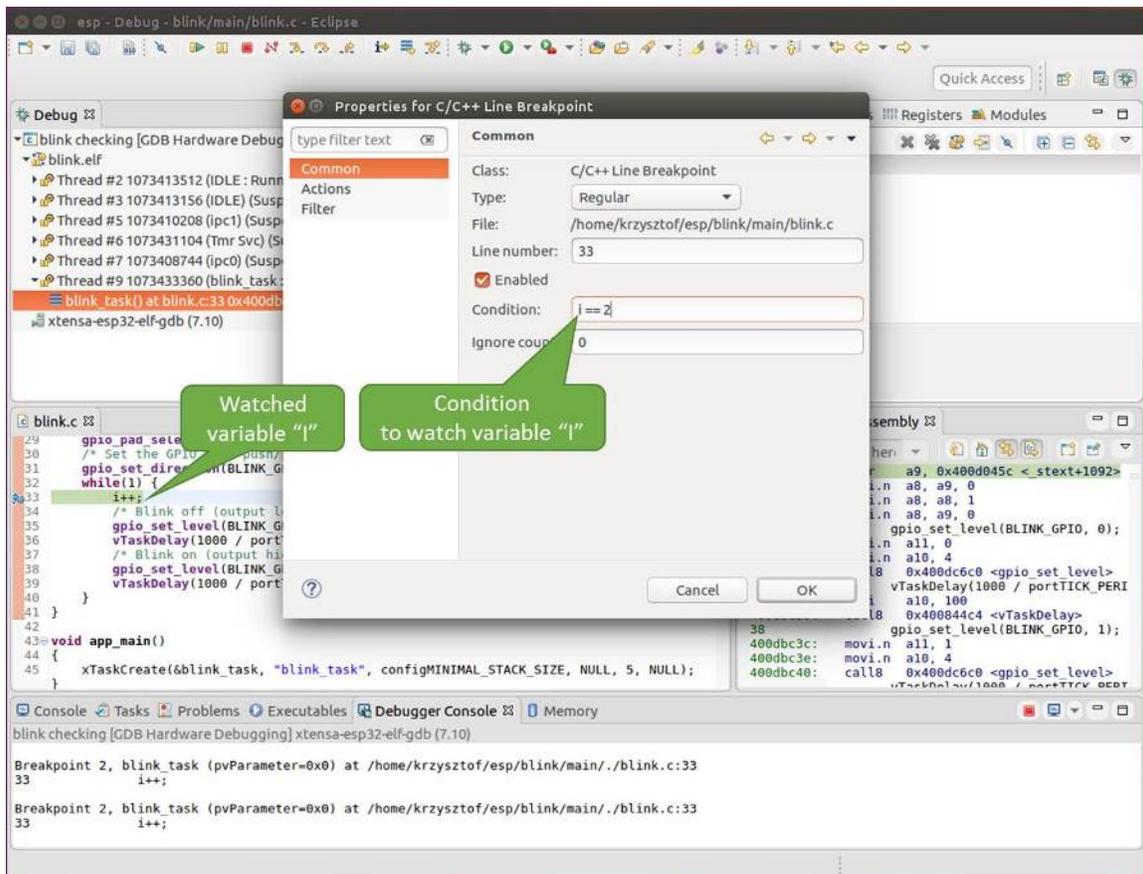


Fig. 21: Setting a conditional breakpoint

## Command Line

Verify if your target is ready and loaded with [get-started/blink](#) example. Configure and start debugger following steps in section *Command Line*. Pick up where target was left by debugger, i.e. having the application halted at breakpoint established at `app_main()`:

```
Temporary breakpoint 1, app_main () at /home/user-name/esp/blink/main/./blink.c:43
43      xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5,
↳NULL);
(gdb)
```

## Examples in this section

1. *Navigating though 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*

## Navigating though the code, call stack and threads

When you see the `(gdb)` prompt, the application is halted. LED should not be blinking.

To find out where exactly the code is halted, enter `l` or `list`, and debugger will show couple of lines of code around the halt point (line 43 of code in file `blink.c`)

```
(gdb) l
38      }
39      }
40
41      void app_main()
42      {
43          xTaskCreate(&blink_task, "blink_task", configMINIMAL_STACK_SIZE, NULL, 5,
↳NULL);
44      }
(gdb)
```

Check how code listing works by entering, e.g. `l 30, 40` to see particular range of lines of code.

You can use `bt` or `backtrace` to see what function calls lead up to this code:

```
(gdb) bt
#0  app_main () at /home/user-name/esp/blink/main/./blink.c:43
#1  0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
↳esp32/./cpu_start.c:339
(gdb)
```

Line #0 of output provides the last function call before the application halted, i.e. `app_main ()` we have listed previously. The `app_main ()` was in turn called by function `main_task` from line 339 of code located in file `cpu_start.c`.

To get to the context of `main_task` in file `cpu_start.c`, enter frame `N`, where `N = 1`, because the `main_task` is listed under #1):

```
(gdb) frame 1
#1 0x400d057e in main_task (args=0x0) at /home/user-name/esp/esp-idf/components/
↳esp32/./cpu_start.c:339
339     app_main();
(gdb)
```

Enter 1 and this will reveal the piece of code that called `app_main ()` (in line 339):

```
(gdb) 1
334         ;
335     }
336 #endif
337     //Enable allocation in region where the startup stacks were located.
338     heap_caps_enable_nonos_stack_heaps();
339     app_main();
340     vTaskDelete(NULL);
341 }
342
(gdb)
```

By listing some lines before, you will see the function name `main_task` we have been looking for:

```
(gdb) 1 326, 341
326     static void main_task(void* args)
327     {
328         // Now that the application is about to start, disable boot watchdogs
329         REG_CLR_BIT(TIMG_WDTCONFIG0_REG(0), TIMG_WDT_FLASHBOOT_MOD_EN_S);
330         REG_CLR_BIT(RTC_CNTL_WDTCONFIG0_REG, RTC_CNTL_WDT_FLASHBOOT_MOD_EN);
331         #if !CONFIG_FREERTOS_UNICORE
332         // Wait for FreeRTOS initialization to finish on APP CPU, before
↳replacing its startup stack
333         while (port_xSchedulerRunning[1] == 0) {
334             ;
335         }
336     #endif
337     //Enable allocation in region where the startup stacks were located.
338     heap_caps_enable_nonos_stack_heaps();
339     app_main();
340     vTaskDelete(NULL);
341 }
(gdb)
```

To see the other code, enter `i threads`. This will show the list of threads running on target:

```
(gdb) i threads
  Id  Target Id  Frame
  8   Thread 1073411336 (dport) 0x400d0848 in dport_access_init_core (arg=<optimized_
↳out>)
    at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:170
  7   Thread 1073408744 (ipc0) xQueueGenericReceive (xQueue=0x3ffae694, pvBuffer=0x0,
↳ xTicksToWait=1644638200,
```

(continues on next page)

(continued from previous page)

```

xJustPeeking=0) at /home/user-name/esp/esp-idf/components/freertos/./queue.c:1452
6   Thread 1073431096 (Tmr Svc) prvTimerTask (pvParameters=0x0)
   at /home/user-name/esp/esp-idf/components/freertos/./timers.c:445
5   Thread 1073410208 (ipcl : Running) 0x4000bfea in ?? ()
4   Thread 1073432224 (dport) dport_access_init_core (arg=0x0)
   at /home/user-name/esp/esp-idf/components/esp32/./dport_access.c:150
3   Thread 1073413156 (IDLE) prvIdleTask (pvParameters=0x0)
   at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
2   Thread 1073413512 (IDLE) prvIdleTask (pvParameters=0x0)
   at /home/user-name/esp/esp-idf/components/freertos/./tasks.c:3282
* 1   Thread 1073411772 (main : Running) app_main () at /home/user-name/esp/blink/
↳main/./blink.c:43
(gdb)

```

The thread list shows the last function calls per each thread together with the name of C source file if available.

You can navigate to specific thread by entering `thread N`, where `N` is the thread Id. To see how it works go to thread 5:

```

(gdb) thread 5
[Switching to thread 5 (Thread 1073410208)]
#0  0x4000bfea in ?? ()
(gdb)

```

Then check the backtrace:

```

(gdb) bt
#0  0x4000bfea in ?? ()
#1  0x40083a85 in vPortCPUReleaseMutex (mux=<optimized out>) at /home/user-name/esp/
↳esp-idf/components/freertos/./port.c:415
#2  0x40083fc8 in vTaskSwitchContext () at /home/user-name/esp/esp-idf/components/
↳freertos/./tasks.c:2846
#3  0x4008532b in _frxt_dispatch ()
#4  0x4008395c in xPortStartScheduler () at /home/user-name/esp/esp-idf/components/
↳freertos/./port.c:222
#5  0x4000000c in ?? ()
#6  0x4000000c in ?? ()
#7  0x4000000c in ?? ()
#8  0x4000000c in ?? ()
(gdb)

```

As you see, the backtrace may contain several entries. This will let you check what exact sequence of function calls lead to the code where the target halted. Question marks `??` instead of a function name indicate that application is available only in binary format, without any source file in C language. The value like `0x4000bfea` is the memory address of the function call.

Using `bt`, `i threads`, `thread N` and `list` commands we are now able to navigate through the code of entire application. This comes handy when stepping though the code and working with breakpoints and will be discussed below.

## Setting and clearing breakpoints

When debugging, we would like to be able to stop the application at critical lines of code and then examine the state of specific variables, memory and registers / peripherals. To do so we are using breakpoints. They provide a convenient way to quickly get to and halt the application at specific line.

Let's establish two breakpoints when the state of LED changes. Basing on code listing above this happens at lines 33 and 36. Breakpoints may be established using command `break M` where `M` is the code line number:

```
(gdb) break 33
Breakpoint 2 at 0x400db6f6: file /home/user-name/esp/blink/main/./blink.c, line 33.
(gdb) break 36
Breakpoint 3 at 0x400db704: file /home/user-name/esp/blink/main/./blink.c, line 36.
```

If you 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:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F6 (active)    APP_CPU: PC=0x400D10D8

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.
↳c:33
33         gpio_set_level(BLINK_GPIO, 0);
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB6F8 (active)    APP_CPU: PC=0x400D10D8
Target halted. PRO_CPU: PC=0x400DB704 (active)    APP_CPU: PC=0x400D10D8

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.
↳c:36
36         gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

You will be also able to see that LED is changing the state only if you resume program execution by entering `c`.

To examine how many breakpoints are set and where, use command `info break`:

```
(gdb) info break
Num      Type          Disp Enb Address          What
2        breakpoint     keep y   0x400db6f6 in blink_task at /home/user-name/esp/blink/
↳main/./blink.c:33
         breakpoint already hit 1 time
3        breakpoint     keep y   0x400db704 in blink_task at /home/user-name/esp/blink/
↳main/./blink.c:36
         breakpoint already hit 1 time
(gdb)
```

Please note that breakpoint numbers (listed under `Num`) start with 2. This is because first breakpoint has been already established at function `app_main()` by running command `thb app_main` on debugger launch. As it was a temporary breakpoint, it has been automatically deleted and now is not listed anymore.

To remove breakpoints enter `delete N` command (in short `d N`), where `N` is the breakpoint number:

```
(gdb) delete 1
No breakpoint number 1.
(gdb) delete 2
(gdb)
```

Read more about breakpoints under [Breakpoints and watchpoints available](#) and [What else should I know about breakpoints?](#)

## Halting and resuming the application

When debugging, you may resume application and enter code waiting for some event or staying in infinite loop without any break points defined. In such case, to go back to debugging mode, you can break program execution manually by entering Ctrl+C.

To check it delete all breakpoints and enter `c` to resume application. Then enter Ctrl+C. Application will be halted at some random point and LED will stop blinking. Debugger will print the following:

```
(gdb) c
Continuing.
^CTarget halted. PRO_CPU: PC=0x400D0C00          APP_CPU: PC=0x400D0C00 (active)
[New Thread 1073433352]

Program received signal SIGINT, Interrupt.
[Switching to Thread 1073413512]
0x400d0c00 in esp_vApplicationIdleHook () at /home/user-name/esp/esp-idf/components/
↳esp32/./freertos_hooks.c:52
52             asm("waiti 0");
(gdb)
```

In particular case above, the application has been halted in line 52 of code in file `freertos_hooks.c`. Now you can resume it again by enter `c` or do some debugging as discussed below.

**Note:** In MSYS2 shell Ctrl+C does not halt the target but exists debugger. To resolve this issue consider debugging with *Eclipse* or check a workaround under [http://www.mingw.org/wiki/Workaround\\_for\\_GDB\\_Ctrl\\_C\\_Interrupt](http://www.mingw.org/wiki/Workaround_for_GDB_Ctrl_C_Interrupt).

## Stepping through the code

It is also possible to step through the code using `step` and `next` commands (in short `s` and `n`). The difference is that `step` is entering inside subroutines calls, while `next` steps over the call, treating it as a single source line.

To demonstrate this functionality, using command `break` and `delete` discussed in previous paragraph, make sure that you have only one breakpoint defined at line 36 of `blink.c`:

```
(gdb) info break
Num      Type           Disp Enb Address      What
3        breakpoint      keep y   0x400db704 in blink_task at /home/user-name/esp/blink/
↳main/./blink.c:36
         breakpoint already hit 1 time
(gdb)
```

Resume program by entering `c` and let it halt:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB754 (active)  APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.
↳c:36
36             gpio_set_level(BLINK_GPIO, 1);
(gdb)
```

Then enter `n` couple of times to see how debugger is stepping one program line at a time:

```
(gdb) n
Target halted. PRO_CPU: PC=0x400DB756 (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB758 (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04C (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active) APP_CPU: PC=0x400D1128
37     vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) n
Target halted. PRO_CPU: PC=0x400DB75E (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400846FC (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB761 (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB746 (active) APP_CPU: PC=0x400D1128
33     gpio_set_level(BLINK_GPIO, 0);
(gdb)
```

If you enter `s` instead, then debugger will step inside subroutine calls:

```
(gdb) s
Target halted. PRO_CPU: PC=0x400DB748 (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB74B (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04C (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DC04F (active) APP_CPU: PC=0x400D1128
gpio_set_level (gpio_num=GPIO_NUM_4, level=0) at /home/user-name/esp/esp-idf/
↳components/driver/./gpio.c:183
183     GPIO_CHECK(GPIO_IS_VALID_OUTPUT_GPIO(gpio_num), "GPIO output gpio_num error",
↳ESP_ERR_INVALID_ARG);
(gdb)
```

In this particular case debugger stepped inside `gpio_set_level(BLINK_GPIO, 0)` and effectively moved to `gpio.c` driver code.

See *Why stepping with “next” does not bypass subroutine calls?* for potential limitation of using `next` command.

## Checking and setting memory

Displaying the contents of memory is done with command `x`. With additional parameters you may vary the format and count of memory locations displayed. Run `help x` to see more details. Companion command to `x` is `set` that let you write values to the memory.

We will demonstrate how `x` and `set` work by reading from and writing to the memory location `0x3FF44004` labeled as `GPIO_OUT_REG` used to set and clear individual GPIO's. For more information please refer to [ESP32 Technical Reference Manual](#), chapter `IO_MUX` and `GPIO Matrix`.

Being in the same `blink.c` project as before, set two breakpoints right after `gpio_set_level` instruction. Enter two times `c` to get to the break point followed by `x /1wx 0x3FF44004` to display contents of `GPIO_OUT_REG` memory location:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB75E (active) APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB74E (active) APP_CPU: PC=0x400D1128

Breakpoint 2, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.
↳c:34
34     vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
```

(continues on next page)

(continued from previous page)

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)    APP_CPU: PC=0x400D1128
Target halted. PRO_CPU: PC=0x400DB75B (active)    APP_CPU: PC=0x400D1128

Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.
↳c:37
37          vTaskDelay(1000 / portTICK_PERIOD_MS);
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000010
(gdb)
```

If you are blinking LED connected to GPIO4, then you should see fourth bit being flipped each time the LED changes the state:

```
0x3ff44004: 0x00000000
...
0x3ff44004: 0x00000010
```

Now, when the LED is off, that corresponds to `0x3ff44004: 0x00000000` being displayed, try using `set` command to set this bit by writing `0x00000010` to the same memory location:

```
(gdb) x /1wx 0x3FF44004
0x3ff44004: 0x00000000
(gdb) set {unsigned int}0x3FF44004=0x000010
```

You should see the LED to turn on immediately after entering `set {unsigned int}0x3FF44004=0x000010` command.

## Watching and setting program variables

A common debugging task is checking the value of a program variable as the program runs. To be able to demonstrate this functionality, update file `blink.c` by adding a declaration of a global variable `int i` above definition of function `blink_task`. Then add `i++` inside `loop(1)` of this function to get `i` incremented on each blink.

Exit debugger, so it is not confused with new code, build and flash the code to the ESP and restart debugger. There is no need to restart OpenOCD.

Once application is halted, enter the command `watch i`:

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb)
```

This will insert so called “watchpoint” in each place of code where variable `i` is being modified. Now enter `continue` to resume the application and observe it being halted:

```
(gdb) c
Continuing.
Target halted. PRO_CPU: PC=0x400DB751 (active)    APP_CPU: PC=0x400D0811
[New Thread 1073432196]

Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 1073432196]
0x400db751 in blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.
↳c:33
```

(continues on next page)

(continued from previous page)

```
33         i++;  
(gdb)
```

Resume application couple more times so `i` gets incremented. Now you can enter `print i` (in short `p i`) to check the current value of `i`:

```
(gdb) p i  
$1 = 3  
(gdb)
```

To modify the value of `i` use `set` command as below (you can then print it out to check if it has been indeed changed):

```
(gdb) set var i = 0  
(gdb) p i  
$3 = 0  
(gdb)
```

You may have up to two watchpoints, see [Breakpoints and watchpoints available](#).

### Setting conditional breakpoints

Here comes more interesting part. You may set a breakpoint to halt the program execution, if certain condition is satisfied. Delete existing breakpoints and try this:

```
(gdb) break blink.c:34 if (i == 2)  
Breakpoint 3 at 0x400db753: file /home/user-name/esp/blink/main/./blink.c, line 34.  
(gdb)
```

Above command sets conditional breakpoint to halt program execution in line 34 of `blink.c` if `i == 2`.

If current value of `i` is less than 2 and program is resumed, it will blink LED in a loop until condition `i == 2` gets true and then finally halt:

```
(gdb) set var i = 0  
(gdb) c  
Continuing.  
Target halted. PRO_CPU: PC=0x400DB755 (active)     APP_CPU: PC=0x400D112C  
Target halted. PRO_CPU: PC=0x400DB753 (active)     APP_CPU: PC=0x400D112C  
Target halted. PRO_CPU: PC=0x400DB755 (active)     APP_CPU: PC=0x400D112C  
Target halted. PRO_CPU: PC=0x400DB753 (active)     APP_CPU: PC=0x400D112C  
  
Breakpoint 3, blink_task (pvParameter=0x0) at /home/user-name/esp/blink/main/./blink.  
↪c:34  
34         gpio_set_level(BLINK_GPIO, 0);  
(gdb)
```

### Obtaining help on commands

Commands presented so far should provide a very basic and intended to let you quickly get started with JTAG debugging. Check help what are the other commands at your disposal. To obtain help on syntax and functionality of 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 <http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>. Good to have as a reference (even if not all commands are applicable in an embedded environment).

## Ending debugger session

To quit debugger enter `q`:

```
(gdb) q
A debugging session is active.

    Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/user-name/esp/blink/build/blink.elf, Remote target
Ending remote debugging.
user-name@computer-name:~/esp/blink$
```

## Tips and Quirks

This section provides collection of all tips and quirks referred to from various parts of this guide.

## Breakpoints and watchpoints available

The ESP32 supports 2 hardware breakpoints. It also supports two watchpoints, so two variables can be watched for change or read by the GDB command `watch myVariable`. Note that menuconfig option `FREERTOS_WATCHPOINT_END_OF_STACK` uses the 2nd watchpoint and will not provide expected results, if you also try to use it within OpenOCD / GDB. See menuconfig's help for detailed description.

## What else should I know about breakpoints?

Normal GDB breakpoints (`b myFunction`) can only be set in IRAM, because that memory is writable. Setting these types of breakpoints in code in flash will not work. Instead, use a hardware breakpoint (`hb myFunction`).

## Why stepping with “next” does not bypass subroutine calls?

When stepping through the code with `next` command, GDB is internally setting a breakpoint (one out of two available) ahead in the code to bypass the subroutine calls. This functionality will not work, if the two available breakpoints are already set elsewhere in the code. If this is the case, delete breakpoints to have one “spare”. With both breakpoints already used, stepping through the code with `next` command will work as like with `step` command and debugger will step inside subroutine calls.

## Support options for OpenOCD at compile time

ESP-IDF has some support options for OpenOCD debugging which can be set at compile time:

- `ESP32_DEBUG_OCDAWARE` is enabled by default. If a panic or unhandled exception is thrown and a JTAG debugger is connected (ie openocd is running), ESP-IDF will break into the debugger.
- `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 `make menuconfig` menu for more details on setting compile-time options.

## FreeRTOS support

OpenOCD has explicit support for the ESP-IDF FreeRTOS. GDB can see FreeRTOS tasks as threads. Viewing them all can be done using the GDB `i threads` command, changing to a certain task is done with `thread n`, with `n` being the number of the thread. FreeRTOS detection can be disabled in target's configuration. For more details see *Configuration of OpenOCD for specific target*.

## Why to set SPI flash voltage in OpenOCD configuration?

The MTDI pin of ESP32, being among four pins used for JTAG communication, is also one of ESP32's bootstrapping pins. On power up ESP32 is sampling binary level on MTDI to set it's internal voltage regulator used to supply power to external SPI flash chip. If binary level on MTDI pin on power up is low, the voltage regulator is set to deliver 3.3V, if it is high, then the voltage is set to 1.8V. The MTDI pin should have a pull-up or may rely on internal weak pull down resistor (see ESP32 Datasheet for details), depending on the type of SPI chip used. Once JTAG is connected, it overrides the pull-up or pull-down resistor that is supposed to do the bootstrapping.

To handle this issue OpenOCD's board configuration file (e.g. `boards\esp-wroom-32.cfg` for ESP-WROOM-32 module) provides `ESP32_FLASH_VOLTAGE` parameter to set the idle state of the TDO line to a specified binary level, therefore reducing the chance of a bad bootup of application due to incorrect flash voltage.

Check specification of ESP32 module connected to JTAG, what is the power supply voltage of SPI flash chip. Then set `ESP32_FLASH_VOLTAGE` accordingly. Most WROOM modules use 3.3V flash, while WROVER modules use 1.8V flash.

## Optimize JTAG speed

In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG. To do so use the following tips.

1. The upper limit of JTAG clock frequency is 20 MHz if CPU runs at 80 MHz, or 26 MHz if CPU runs at 160 MHz or 240 MHz.
2. Depending on particular JTAG adapter and the length of connecting cables, you may need to reduce JTAG frequency below 20 / 26 MHz.
3. In particular reduce frequency, if you get DSR/DIR errors (and they do not relate to OpenOCD trying to read from a memory range without physical memory being present there).
4. ESP-WROVER-KIT operates stable at 20 / 26 MHz.

## What is the meaning of debugger's startup commands?

On startup, debugger is issuing sequence of commands to reset the chip and halt it at specific line of code. This sequence (shown below) is user defined to pick up at most convenient / appropriate line and start debugging.

- `mon reset halt` — reset the chip and keep the CPUs halted
- `thb app_main` — insert a temporary hardware breakpoint at `app_main`, put here another function name if required
- `x $a1=0` — this is the tricky part. As far as we can tell, there is no way for a `mon` command to tell 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 `x $a1=0` is a way to force GDB to get new state from the target.
- `c` — resume the program. It will then stop at breakpoint inserted at `app_main`.

## Configuration of OpenOCD for specific target

OpenOCD needs to be told what JTAG adapter **interface** to use, as well as what type of **board** and processor the JTAG adapter is connected to. To do so, use existing configuration files located in OpenOCD's `share/openocd/scripts/interface` and `share/openocd/scripts/board` folders.

For example, if you connect to ESP-WROVER-KIT with ESP-WROOM-32 module installed (see section [ESP32 WROVER KIT](#)), use the following configuration files:

- `interface/ftdi/esp32_devkitj_v1.cfg`
- `board/esp-wroom-32.cfg`

Optionally prepare configuration by yourself. To do so, you can check existing files and modify them to match you specific hardware. Below is the summary of available configuration parameters for **board** configuration.

### Adapter's clock speed

```
adapter_khz 20000
```

See [Optimize JTAG speed](#) for guidance how to set this value.

### Single core debugging

```
set ESP32_ONLYCPU 1
```

Comment out this line for dual core debugging.

### Disable RTOS support

```
set ESP32_RTOS none
```

Comment out this line to have RTOS support.

## Power supply voltage of ESP32's SPI flash chip

```
set ESP32_FLASH_VOLTAGE 1.8
```

Comment out this line to set 3.3V, ref: *Why to set SPI flash voltage in OpenOCD configuration?*

## Configuration file for ESP32 targets

```
source [find target/esp32.cfg]
```

---

**Note:** Do not change source [find target/esp32.cfg] line unless you are familiar with OpenOCD internals.

---

Currently target/esp32.cfg remains the only configuration file for ESP32 targets (esp108 and esp32). The matrix of supported configurations is as follows:

Dual/single	RTOS	Target used
dual	FreeRTOS	esp32
single	FreeRTOS	esp108 (*)
dual	none	esp108
single	none	esp108

(\*) — we plan to fix this and add support for single core debugging with esp32 target in a subsequent commits.

Look inside board/esp-wroom-32.cfg for additional information provided in comments besides each configuration parameter.

## How debugger resets ESP32?

The board can be reset by entering `mon reset` or `mon reset halt` into GDB.

## Do not use JTAG pins for something else

Operation of JTAG may be disturbed, if some other h/w is connected to JTAG pins besides ESP32 module and JTAG adapter. ESP32 JTAG us using the following pins:

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

JTAG communication will likely fail, if configuration of JTAG pins is changed by user application. If OpenOCD initializes correctly (detects the two Tensilica cores), but loses sync and spews out a lot of DTR/DIR errors when the program is ran, it is likely that the application reconfigures the JTAG pins to something else, or the user forgot to connect Vtar to a JTAG adapter that needed it.

Below is an excerpt from series of errors reported by GDB after the application stepped into the code that reconfigured MTDO / GPIO15 to be an input:

```
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates target still busy!
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated an
↳exception!
cpu0: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated an
↳overrun!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates target still busy!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated an
↳exception!
cpu1: xtensa_resume (line 431): DSR (FFFFFFFF) indicates DIR instruction generated an
↳overrun!
```

## Reporting issues with OpenOCD / GDB

In case you encounter a problem with OpenOCD or GDB programs itself and do not find a solution searching available resources on the web, open an issue in the OpenOCD issue tracker under <https://github.com/espressif/openocd-esp32/issues>.

1. In issue report provide details of your configuration:
  - a. JTAG adapter type.
  - b. Release of ESP-IDF used to compile and load application that is being debugged.
  - c. Details of OS used for debugging.
  - d. Is OS running natively on a PC or on a virtual machine?
2. Create a simple example that is representative to observed issue. Describe steps how to reproduce it. In such an example debugging should not be affected by non-deterministic behaviour introduced by the Wi-Fi stack, so problems will likely be easier to reproduce, if encountered once.
3. Prepare logs from debugging session by adding additional parameters to start up commands.

OpenOCD:

```
bin/openocd -l openocd_log.txt -d 3 -s share/openocd/scripts -f interface/
↳ftdi/esp32_devkitj_v1.cfg -f board/esp-wroom-32.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 `-d 3` is set. If you still like to see the log on the screen, then use another command instead:

```
bin/openocd -d 3 -s share/openocd/scripts -f interface/ftdi/esp32_devkitj_
↳v1.cfg -f board/esp-wroom-32.cfg 2>&1 | tee openocd.log
```

---

**Note:** See *Building OpenOCD from Sources* for slightly different command format, when running OpenOCD built from sources.

---

Debugger:

```
xtensa-esp32-elf-gdb -ex "set remotelogfile gdb_log.txt" <all other_
↳options>
```

Optionally add command `remotelogfile gdb_log.txt` to the `gdbinit` file.

4. Attach both `openocd_log.txt` and `gdb_log.txt` files to your issue report.

## Application Level Tracing library

### Overview

IDF provides useful feature for program behaviour analysis: application level tracing. It is implemented in the corresponding library and can be enabled in `menuconfig`. This feature allows to transfer arbitrary data between host and ESP32 via JTAG interface with small overhead on program execution.

Developers can use this library to send application specific state of execution to the host and receive commands or other type of info in the opposite direction at runtime. The main use cases of this library are:

1. Collecting application specific data, see [Application Specific Tracing](#)
2. Lightweight logging to the host, see [Logging to Host](#)
3. System behaviour analysis, see [System Behaviour Analysis with SEGGER SystemView](#)

Tracing components when working over JTAG interface are shown in the figure below.

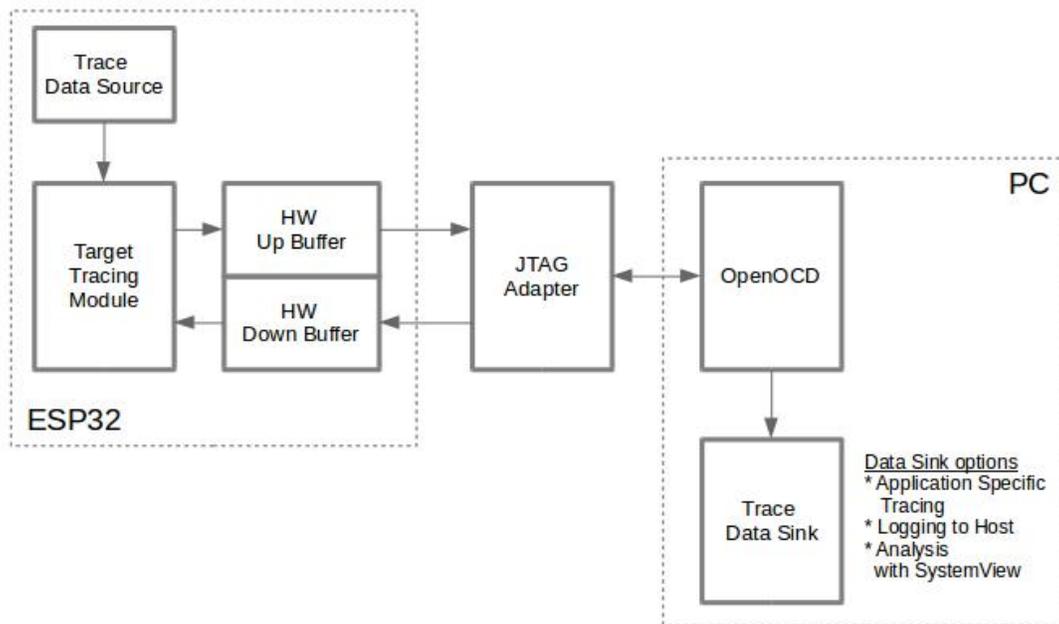


Fig. 22: Tracing Components when Working Over JTAG

### Modes of Operation

The library supports two modes of operation:

**Post-mortem mode.** This is the default mode. The mode does not need interaction from the host side. In this mode tracing module does not check whether host has read all the data from `HW UP BUFFER` buffer and overwrites old data

with the new ones. This mode is useful when only the latest trace data are interesting to the user, e.g. for analyzing program's behaviour just before the crash. Host can read the data later on upon user request, e.g. via special OpenOCD command in case of working via JTAG interface.

**Streaming mode.** Tracing module enters this mode when host connects to ESP32. In this mode before writing new data to *HW UP BUFFER* tracing module checks that there is enough space in it and if necessary waits for the host to read data and free enough memory. Maximum waiting time is controlled via timeout values passed by users to corresponding API routines. So when application tries to write data to trace buffer using finite value of the maximum waiting time it is possible situation that this data will be dropped. Especially this is true for tracing from time critical code (ISRs, OS scheduler code etc.) when infinite timeouts can lead to system malfunction. In order to avoid loss of such critical data developers can enable additional data buffering via menuconfig option *ESP32\_APPTRACE\_PENDING\_DATA\_SIZE\_MAX*. This macro specifies the size of data which can be buffered in above conditions. The option can also help to overcome situation when data transfer to the host is temporarily slowed down, e.g. due to USB bus congestions etc. But it will not help when average bitrate of trace data stream exceeds HW interface capabilities.

## 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 how to set it up, please, see *JTAG Debugging* for details.
2. **Target side:** Application tracing functionality can be enabled in menuconfig. *Component config > Application Level Tracing* menu allows selecting destination for the trace data (HW interface for transport). Choosing any of the destinations automatically enables `CONFIG_ESP32_APPTRACE_ENABLE` option.

---

**Note:** In order to achieve higher data rates and minimize number of dropped packets it is recommended to optimize setting of JTAG clock frequency, so it is at maximum and still provides stable operation of JTAG, see *Optimize JTAG speed*.

---

There are two additional menuconfig options not mentioned above:

1. *Threshold for flushing last trace data to host on panic (ESP32\_APPTRACE\_POSTMORTEM\_FLUSH\_TRAX\_THRESH).*  
This option is necessary due to the nature of working over JTAG. In that mode trace data are exposed to the host in 16KB blocks. In post-mortem mode when one block is filled it is exposed to the host and the previous one becomes unavailable. In other words trace data are overwritten in 16KB granularity. On panic the latest data from the current input block are exposed to host and host can read them for post-analysis. It can happen that system panic occurs when there are very small amount of data which are not exposed to the host yet. In this case the previous 16KB of collected data will be lost and host will see the latest, but very small piece of the trace. It can be insufficient to diagnose the problem. This menuconfig option allows avoiding such situations. It controls the threshold for flushing data in case of panic. For example user can decide that it needs not less then 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 previous 16KB. 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 (ESP32\_APPTRACE\_ONPANIC\_HOST\_FLUSH\_TMO).*  
The option is only meaningful in streaming mode and controls the maximum time tracing module will wait for the host to read the last data in case of panic.

## How to use this library

This library provides API for transferring arbitrary data between host and ESP32. When enabled in menuconfig target application tracing module is initialized automatically at the system startup, so all what the user needs to do is to call

corresponding API to send, receive or flush the data.

## Application Specific Tracing

In general user should decide what type of data should be transferred in every direction and how these data must be interpreted (processed). The following steps must be performed to transfer data between target and host:

1. On target side user should implement algorithms for writing trace data to the host. Piece of code below shows an example how to do this.

```
#include "esp_app_trace.h"
...
char buf[] = "Hello World!";
esp_err_t res = esp_apptrace_write(ESP_APPTRACE_DEST_TRAX, buf, strlen(buf), ESP_
↳APPTRACE_TMO_INFINITE);
if (res != ESP_OK) {
    ESP_LOGE(TAG, "Failed to write data to host!");
    return res;
}
```

esp\_apptrace\_write() function uses memcpy to copy user data to the internal buffer. In some cases it can be more optimal to use esp\_apptrace\_buffer\_get() and esp\_apptrace\_buffer\_put() functions. They allow developers to allocate buffer and fill it themselves. The following piece of code shows how to do this.

```
#include "esp_app_trace.h"
...
int number = 10;
char *ptr = (char *)esp_apptrace_buffer_get(ESP_APPTRACE_DEST_TRAX, 32, 100/*tmo_
↳in us*/);
if (ptr == NULL) {
    ESP_LOGE("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("Failed to put buffer!");
    return res;
}
```

Also according to his needs user may want to receive data from the host. Piece of code below shows an example how to do this.

```
#include "esp_app_trace.h"
...
char buf[32];
char down_buf[32];
size_t sz = sizeof(buf);

/* config down buffer */
esp_apptrace_down_buffer_config(down_buf, sizeof(down_buf));
/* check for incoming data and read them if any */
```

(continues on next page)

(continued from previous page)

```

esp_err_t res = esp_appttrace_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 */
    ...
}

```

esp\_appttrace\_read() function uses memcpy to copy host data to user buffer. In some cases it can be more optimal to use esp\_appttrace\_down\_buffer\_get() and esp\_appttrace\_down\_buffer\_put() functions. They allow developers to occupy chunk of read buffer and process it in-place. The following piece of code shows how to do this.

```

#include "esp_app_trace.h"
...
char down_buf[32];
uint32_t *number;
size_t sz = 32;

/* config down buffer */
esp_appttrace_down_buffer_config(down_buf, sizeof(down_buf));
char *ptr = (char *)esp_appttrace_down_buffer_get(ESP_APPTRACE_DEST_TRAX, &sz, 100/
↳tmo in us*/);
if (ptr == NULL) {
    ESP_LOGE("Failed to get buffer!");
    return ESP_FAIL;
}
if (sz > 4) {
    number = (uint32_t *)ptr;
    printf("Here is the number %d", *number);
} else {
    printf("No data");
}
esp_err_t res = esp_appttrace_down_buffer_put(ESP_APPTRACE_DEST_TRAX, ptr, 100/
↳tmo in us*/);
if (res != ESP_OK) {
    /* in case of error host tracing tool (e.g. OpenOCD) will report incomplete_
↳user buffer */
    ESP_LOGE("Failed to put buffer!");
    return res;
}

```

2. The next step is to build the program image and download it to the target as described in *Build and Flash*.
3. Run OpenOCD (see *JTAG Debugging*).
4. Connect to OpenOCD telnet server. It can be done using the following command in terminal telnet <oocd\_host> 4444. If telnet session is opened on the same machine which runs OpenOCD you can use localhost as <oocd\_host> in the command above.
5. Start trace data collection using special OpenOCD command. This command will transfer tracing data and redirect them to specified file or socket (currently only files are supported as trace data destination). For description of the corresponding commands see *OpenOCD Application Level Tracing Commands*.
6. The final step is to process received data. Since format of data is defined by user the processing stage is out

of the scope of this document. Good starting points for data processor are python scripts in `$IDF_PATH/tools/esp_app_trace`: `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 filling of the allocated memory is performed on behalf of the API caller (in task or ISR context). In multithreading environment it can happen that task/ISR which fills the buffer is preempted by another high priority task/ISR. So it is possible situation that user data preparation process is not completed at the moment when that chunk is read by the host. To handle such conditions tracing module prepends all user data chunks with header which contains allocated user buffer size (2 bytes) and length of actually written data (2 bytes). So total length of the header is 4 bytes. OpenOCD command which reads trace data reports error when it reads incomplete user data chunk, but in any case it puts contents of the whole user chunk (including unfilled area) to output file.

Below is the description of available OpenOCD application tracing commands.

---

**Note:** Currently OpenOCD does not provide commands to send arbitrary user data to the target.

---

Command usage:

```
esp32 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 a file “trace.log”. The file will be saved in “openocd-esp32” directory.

```
esp32 appttrace start file://trace.log 1 2048 5 0 0
```

The tracing data will be retrieved and saved in non-blocking mode. This process will stop automatically after 2048 bytes are collected, or if no data are available for more than 5 seconds.

---

**Note:** Tracing data is buffered before it is made available to OpenOCD. If you see “Data timeout!” message, then the target is likely sending not enough data to empty the buffer to OpenOCD before expiration of timeout. Either increase the timeout or use a function `esp_appttrace_flush()` to flush the data on specific intervals.

---

2. Retrieve tracing data indefinitely in non-blocking mode.

```
esp32 appttrace start file://trace.log 1 -1 -1 0 0
```

There is no limitation on the size of collected data and there is no any data timeout set. This process may be stopped by issuing `esp32 appttrace stop` command on OpenOCD telnet prompt, or by pressing Ctrl+C in OpenOCD window.

3. Retrieve tracing data and save them indefinitely.

```
esp32 appttrace start file://trace.log 0 -1 -1 0 0
```

OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing press Ctrl+C in OpenOCD window.

4. Wait for target to be halted. Then resume target’s operation and start data retrieval. Stop after collecting 2048 bytes of data:

```
esp32 appttrace start file://trace.log 0 2048 -1 1 0
```

To configure tracing immediately after reset use the `openocd reset halt` command.

## Logging to Host

IDF implements useful feature: logging to host via application level tracing library. This is a kind of semihosting when all `ESP_LOGx` calls sends strings to be printed to the host instead of UART. This can be useful because “printing to host” eliminates some steps performed when logging to UART. The most part of work is done on the host.

By default IDF’s logging library uses `vprintf`-like function to write formatted output to dedicated UART. In general it involves the following steps:

1. Format string is parsed to obtain type of each argument.
2. According to its type every argument is converted to string representation.
3. Format string combined with converted arguments is sent to UART.

Though implementation of `vprintf`-like function can be optimised 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 is frequent situation when addition of extra logging to the program to diagnose some problem changes its behaviour and problem disappears or in the worst cases program can not work normally at all and ends up with an error or even hangs.

Possible ways to overcome this problem are to use higher UART bitrates (or another faster interface) and/or move string formatting procedure to the host.

Application level tracing feature can be used to transfer log information to host using `esp_apptrace_vprintf` function. This function does not perform full parsing of the format string and arguments, instead it just calculates number of arguments passed and sends them along with the format string address to the host. On the host log data are processed and printed out by a special Python script.

### Limitations

Current implementation of logging over JTAG has some limitations:

1. Tracing from `ESP_EARLY_LOGx` macros is not supported.
2. No support for `printf` arguments which size exceeds 4 bytes (e.g. `double` and `uint64_t`).
3. Only strings from `.rodata` section are supported as format strings and arguments.
4. Maximum number of `printf` arguments is 256.

### How To Use It

In order to use logging via trace module user needs to perform the following steps:

1. On target side special `vprintf`-like function needs to be installed. As it was mentioned earlier this function is `esp_apptrace_vprintf`. It sends log data to the host. Example code is provided in [system/app\\_trace\\_to\\_host](#).
2. Follow instructions in items 2-5 in *Application Specific Tracing*.
3. To print out collected log records, run the following command in terminal: `$IDF_PATH/tools/esp_app_trace/logtrace_proc.py /path/to/trace/file /path/to/program/elf/file`.

### Log Trace Processor Command Options

Command usage:

```
logtrace_proc.py [-h] [--no-errors] <trace_file> <elf_file>
```

Positional arguments:

**trace\_file** Path to log trace file

**elf\_file** Path to program ELF file

Optional arguments:

**-h, --help** show this help message and exit

**--no-errors, -n** Do not print errors

## System Behaviour Analysis with SEGGER SystemView

Another useful IDF feature built on top of application tracing library is the system level tracing which produces traces compatible with SEGGER SystemView tool (see [SystemView](#)). SEGGER SystemView is a real-time recording and visualization tool that allows to analyze runtime behavior of an application.

---

**Note:** Currently IDF-based application is able to generate SystemView compatible traces in form of files to be opened in SystemView application. The tracing process can not yet be controlled using that tool.

---

### How To Use It

Support for this feature is enabled by *Component config > Application Level Tracing > FreeRTOS SystemView Tracing (SYSVIEW\_ENABLE)* menuconfig option. There are several other options enabled under the same menu:

1. *ESP32 timer to use as SystemView timestamp source (SYSVIEW\_TS\_SOURCE)* selects the source of timestamps for SystemView events. In single core mode timestamps are generated using ESP32 internal cycle counter running at maximum 240 Mhz (~4 ns granularity). In dual-core mode external timer working at 40Mhz is used, so timestamp granularity is 25 ns.
2. Individually enabled or disabled collection of SystemView events (`CONFIG_SYSVIEW_EVT_XXX`):
  - Trace Buffer Overflow Event
  - ISR Enter Event
  - ISR Exit Event
  - ISR Exit to Scheduler Event
  - Task Start Execution Event
  - Task Stop Execution Event
  - Task Start Ready State Event
  - Task Stop Ready State Event
  - Task Create Event
  - Task Terminate Event
  - System Idle Event
  - Timer Enter Event
  - Timer Exit Event

IDF has all the code required to produce SystemView compatible traces, so user can just configure necessary project options (see above), build, download the image to target and use OpenOCD to collect data as described in the previous sections.

### OpenOCD SystemView Tracing Command Options

Command usage:

```
esp32 sysview [start <options>] | [stop] | [status]
```

Sub-commands:

**start** Start tracing (continuous streaming).

**stop** Stop tracing.

**status** Get tracing status.

Start command syntax:

```
start <outfile1> [outfile2] [poll_period [trace_size [stop_tmo]]]
```

**outfile1** Path to file to save data from PRO CPU. This argument should have the following format: `file://path/to/file`.

**outfile2** Path to file to save data from APP CPU. This argument should have the following format: `file://path/to/file`.

**poll\_period** Data polling period (in ms) for available trace data. If greater than 0 then command runs in non-blocking mode. By default 1 ms.

**trace\_size** Maximum size of data to collect (in bytes). Tracing is stopped after specified amount of data is received. By default -1 (trace size stop trigger is disabled).

**stop\_tmo** Idle timeout (in sec). Tracing is stopped if there is no data for specified period of time. By default -1 (disable this stop trigger).

---

**Note:** If `poll_period` is 0 OpenOCD telnet command line will not be available until tracing is stopped. You must stop it manually by resetting the board or pressing Ctrl+C in OpenOCD window (not one with the telnet session). Another option is to set `trace_size` and wait until this size of data is collected. At this point tracing stops automatically.

---

Command usage examples:

1. Collect SystemView tracing data to files “pro-cpu.SVdat” and “app-cpu.SVdat”. The files will be saved in “openocd-esp32” directory.

```
esp32 sysview start file://pro-cpu.SVdat file://app-cpu.SVdat
```

The tracing data will be retrieved and saved in non-blocking mode. To stop data this process enter `esp32 apptrace stop` command on OpenOCD telnet prompt, Optionally pressing Ctrl+C in OpenOCD window.

2. Retrieve tracing data and save them indefinitely.

```
esp32 sysview start file://pro-cpu.SVdat file://app-cpu.SVdat 0 -1 -1
```

OpenOCD telnet command line prompt will not be available until tracing is stopped. To stop tracing, press Ctrl+C in OpenOCD window.

## Data Visualization

After trace data are collected user can use special tool to visualize the results and inspect behaviour of the program. Unfortunately SystemView does not support tracing from multiple cores. So when tracing from ESP32 working in dual-core mode two files are generated: one for PRO CPU and another one for APP CPU. User can load every file into separate instance of the tool.

It is uneasy and awkward to analyze data for every core in separate instance of the tool. Fortunately there is Eclipse plugin called *Impulse* which can load several trace files and makes it possible to inspect events from both cores in one view. Also this plugin has no limitation of 1000000 events as compared to free version of SystemView.

Good instruction on how to install, configure and visualize data in Impulse from one core can be found [here](#).

**Note:** IDF uses its own mapping for SystemView FreeRTOS events IDs, so user needs to replace original file with mapping `$(SYSVIEW_INSTALL_DIR)/Description/SYSVIEW_FreeRTOS.txt` with `$(IDF_PATH)/docs/api-guides/SYSVIEW_FreeRTOS.txt`. Also contents of that IDF specific file should be used when configuring SystemView serializer using above link.

---

## Configure Impulse for Dual Core Traces

After installing Impulse and ensuring that it can successfully load trace files for each core in separate tabs user can add special Multi Adapter port and load both files into one view. To do this user needs to do the following in Eclipse:

1. Open 'Signal Ports' view. Go to Windows->Show View->Other menu. Find 'Signal Ports' view in Impulse folder and double-click on it.
2. In 'Signal Ports' view right-click on 'Ports' and select 'Add ...'->New Multi Adapter Port
3. In open dialog Press 'Add' button and select 'New Pipe/File'.
4. In open dialog select 'SystemView Serializer' as Serializer and set path to PRO CPU trace file. Press OK.
5. Repeat steps 3-4 for APP CPU trace file.
6. Double-click on created port. View for this port should open.
7. Click Start/Stop Streaming button. Data should be loaded.
8. Use 'Zoom Out', 'Zoom In' and 'Zoom Fit' button to inspect data.
9. For settings measurement cursors and other features please see [Impulse documentation](#)).

**Note:** If you have problems with visualization (no data are shown or strange behaviour of zoom action is observed) you can try to delete current signal hierarchy and double click on necessary file or port. Eclipse will ask you to create new signal hierarchy.

---

## 4.9 Partition Tables

### 4.9.1 Overview

A single ESP32's flash can contain multiple apps, as well as many different kinds of data (calibration data, filesystems, parameter storage, etc). For this reason a partition table is flashed to offset 0x8000 in the flash.

Partition table length is 0xC00 bytes (maximum 95 partition table entries). If the partition table is signed due to *secure boot*, the signature is appended after the table data.

Each entry in the partition table has a name (label), type (app, data, or something else), subtype and the offset in flash where the partition is loaded.

The simplest way to use the partition table is to *make menuconfig* and choose one of the simple predefined partition tables:

- "Single factory app, no OTA"
- "Factory app, two OTA definitions"

In both cases the factory app is flashed at offset 0x10000. If you *make partition\_table* then it will print a summary of the partition table.

## 4.9.2 Built-in Partition Tables

Here is the summary printed for the “Single factory app, no OTA” configuration:

```
# Espressif ESP32 Partition Table
# Name, Type, SubType, Offset, Size
nvs, data, nvs, 0x9000, 0x6000
phy_init, data, phy, 0xf000, 0x1000
factory, app, factory, 0x10000, 1M
```

- At a 0x10000 (64KB) offset in the flash is the app labelled “factory”. The bootloader will run this app by default.
- There are also two data regions defined in the partition table for storing NVS library partition and PHY init data.

Here is the summary printed for the “Factory app, two OTA definitions” configuration:

```
# Espressif ESP32 Partition Table
# Name, Type, SubType, Offset, Size
nvs, data, nvs, 0x9000, 0x4000
otadata, data, ota, 0xd000, 0x2000
phy_init, data, phy, 0xf000, 0x1000
factory, 0, 0, 0x10000, 1M
ota_0, 0, ota_0, , 1M
ota_1, 0, ota_1, , 1M
```

- There are now three app partition definitions.
- The type of all three are set as “app”, but the subtype varies between the factory app at 0x10000 and the next two “OTA” apps.
- There is also a new “ota data” 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.

## 4.9.3 Creating Custom Tables

If you choose “Custom partition table CSV” in menuconfig then you can also enter the name of a CSV file (in the project directory) to use for your partition table. The CSV file can describe any number of definitions for the table you need.

The CSV format is the same format as printed in the summaries shown above. However, not all fields are required in the CSV. For example, here is the “input” CSV for the OTA partition table:

```
# Name, Type, SubType, Offset, Size
nvs, data, nvs, 0x9000, 0x4000
otadata, data, ota, 0xd000, 0x2000
phy_init, data, phy, 0xf000, 0x1000
factory, app, factory, 0x10000, 1M
ota_0, app, ota_0, , 1M
ota_1, app, ota_1, , 1M
```

- 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.
- Only the offset for the first partition is supplied. The `gen_esp32part.py` tool fills in each remaining offset to start after the preceding partition.

## Name field

Name field can be any meaningful name. It is not significant to the ESP32. Names longer than 16 characters will be truncated.

## Type field

Partition type field can be specified as app (0) or data (1). Or it can be a number 0-254 (or as hex 0x00-0xFE). Types 0x00-0x3F are reserved for esp-idf core functions.

If your application needs to store data, please add a custom partition type in the range 0x40-0xFE.

The bootloader ignores any partition types other than app (0) & data (1).

## Subtype

The 8-bit subtype field is specific to a given partition type.

esp-idf currently only specifies the meaning of the subtype field for “app” and “data” partition types.

## App Subtypes

When type is “app”, the subtype field can be specified as factory (0), ota\_0 (0x10) ... ota\_15 (0x1F) or test (0x20).

- factory (0) is the default app partition. The bootloader will execute the factory app unless there it sees a partition of type data/ota, in which case it reads this partition to determine which OTA image to boot.
  - OTA never updates the factory partition.
  - If you want to conserve flash usage in an OTA project, you can remove the factory partition and use ota\_0 instead.
- ota\_0 (0x10) ... ota\_15 (0x1F) are the OTA app slots. Refer to the [OTA documentation](#) for more details, which then use the OTA data partition to configure which app slot the bootloader should boot. If using OTA, an application should have at least two OTA application slots (ota\_0 & ota\_1). Refer to the [OTA documentation](#) for more details.
- test (0x2) is a reserved subtype for factory test procedures. It is not currently supported by the esp-idf bootloader.

## Data Subtypes

When type is “data”, the subtype field can be specified as ota (0), phy (1), nvs (2).

- ota (0) is the [OTA data partition](#) which stores information about the currently selected OTA application. This partition should be 0x2000 bytes in size. Refer to the [OTA documentation](#) for more details.
- phy (1) is for storing PHY initialisation data. This allows PHY to be configured per-device, instead of in firmware.
  - In the default configuration, the phy partition is not used and PHY initialisation data is compiled into the app itself. As such, this partition can be removed from the partition table to save space.
  - To load PHY data from this partition, run `make menuconfig` and enable [ESP32\\_PHY\\_INIT\\_DATA\\_IN\\_PARTITION](#) option. You will also need to flash your devices with phy init data as the esp-idf build system does not do this automatically.
- nvs (2) is for the [Non-Volatile Storage \(NVS\) API](#).

- NVS is used to store per-device PHY calibration data (different to initialisation data).
- NVS is used to store WiFi data if the `esp_wifi_set_storage(WIFI_STORAGE_FLASH)` initialisation function is used.
- The NVS API can also be used for other application data.
- It is strongly recommended that you include an NVS partition of at least 0x3000 bytes in your project.
- If using NVS API to store a lot of data, increase the NVS partition size from the default 0x6000 bytes.

Other data subtypes are reserved for future esp-idf uses.

## Offset & Size

Only the first offset field is required (we recommend using 0x10000). Partitions with blank offsets will start after the previous partition.

App partitions have to be at offsets aligned to 0x10000 (64K). If you leave the offset field blank, the tool 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).

### 4.9.4 Generating Binary Partition Table

The partition table which is flashed to the ESP32 is in a binary format, not CSV. The tool `partition_table/gen_esp32part.py` is used to convert between CSV and binary formats.

If you configure the partition table CSV name in `make menuconfig` and then `make partition_table`, this conversion is done as part of the build process.

To convert CSV to Binary manually:

```
python gen_esp32part.py --verify input_partitions.csv binary_partitions.bin
```

To convert binary format back to CSV:

```
python gen_esp32part.py --verify 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 `make partition_table` are generated:

```
python gen_esp32part.py binary_partitions.bin
```

`gen_esp32part.py` takes one optional argument, `--verify`, which will also verify the partition table during conversion (checking for overlapping partitions, unaligned partitions, etc.)

### 4.9.5 Flashing the partition table

- `make partition_table-flash`: will flash the partition table with `esptool.py`.
- `make flash`: Will flash everything including the partition table.

A manual flashing command is also printed as part of `make partition_table`.

Note that updating the partition table doesn't erase data that may have been stored according to the old partition table. You can use `make erase_flash` (or `esptool.py erase_flash`) to erase the entire flash contents.

## 4.10 Secure Boot

Secure Boot is a feature for ensuring only your code can run on the chip. Data loaded from flash is verified on each reset.

Secure Boot is separate from the *Flash Encryption* feature, and you can use secure boot without encrypting the flash contents. However we recommend using both features together for a secure environment. See *Secure Boot & Flash Encryption* for more details.

---

**Important:** Enabling secure boot limits your options for further updates of your ESP32. Make sure to read this document thoroughly and understand the implications of enabling secure boot.

---

### 4.10.1 Background

- Most data is stored in flash. Flash access does not need to be protected from physical access in order for secure boot to function, because critical data is stored (non-software-accessible) in Efuses internal to the chip.
- Efuses are used to store the secure bootloader key (in efuse BLOCK2), and also a single Efuse bit (ABS\_DONE\_0) is burned (written to 1) to permanently enable secure boot on the chip. For more details about efuse, see Chapter 11 “eFuse Controller” in the Technical Reference Manual.
- To understand the secure boot process, first familiarise yourself with the standard *ESP-IDF boot process*.
- Both stages of the boot process (initial software bootloader load, and subsequent partition & app loading) are verified by the secure boot process, in a “chain of trust” relationship.

### 4.10.2 Secure Boot Process Overview

This is a high level overview of the secure boot process. Step by step instructions are supplied under *How To Enable Secure Boot*. Further in-depth details are supplied under *Technical Details*:

1. The options to enable secure boot are provided in the `make menuconfig` hierarchy, under “Secure Boot Configuration”.
2. Secure Boot defaults to signing images and partition table data during the build process. The “Secure boot private signing key” config item is a file path to a ECDSA public/private key pair in a PEM format file.
3. The software bootloader image is built by `esp-idf` with secure boot support enabled and the public key (signature verification) portion of the secure boot signing key compiled in. This software bootloader image is flashed at offset 0x1000.
4. On first boot, the software bootloader follows the following process to enable secure boot:
  - Hardware secure boot support generates a device secure bootloader key (generated via hardware RNG, then stored read/write protected in efuse), and a secure digest. The digest is derived from the key, an IV, and the bootloader image contents.
  - The secure digest is flashed at offset 0x0 in the flash.
  - Depending on Secure Boot Configuration, efuses are burned to disable JTAG and the ROM BASIC interpreter (it is strongly recommended these options are turned on.)
  - Bootloader permanently enables secure boot by burning the ABS\_DONE\_0 efuse. The software bootloader then becomes protected (the chip will only boot a bootloader image if the digest matches.)

5. On subsequent boots the ROM bootloader sees that the secure boot efuse is burned, reads the saved digest at 0x0 and uses hardware secure boot support to compare it with a newly calculated digest. If the digest does not match then booting will not continue. The digest and comparison are performed entirely by hardware, and the calculated digest is not readable by software. For technical details see *Secure Boot Hardware Support*.
6. When running in secure boot mode, the software bootloader uses the secure boot signing key (the public key of which is embedded in the bootloader itself, and therefore validated as part of the bootloader) to verify the signature appended to all subsequent partition tables and app images before they are booted.

### 4.10.3 Keys

The following keys are used by the secure boot process:

- “secure bootloader key” is a 256-bit AES key that is stored in Efuse block 2. The bootloader can generate this key itself from the internal hardware random number generator, the user does not need to supply it (it is optionally possible to supply this key, see *Re-Flashable Software Bootloader*). The Efuse holding this key is read & write protected (preventing software access) before secure boot is enabled.
- “secure boot signing key” is a standard ECDSA public/private key pair (see *Image Signing Algorithm*) in PEM format.
  - The public key from this key pair (for signature verification but not signature creation) is compiled into the software bootloader and used to verify the second stage of booting (partition table, app image) before booting continues. The public key can be freely distributed, it does not need to be kept secret.
  - The private key from this key pair *must be securely kept private*, as anyone who has this key can authenticate to any bootloader that is configured with secure boot and the matching public key.

### 4.10.4 How To Enable Secure Boot

1. Run `make menuconfig`, navigate to “Secure Boot Configuration” and select the option “One-time Flash”. (To understand the alternative “Reflashable” choice, see *Re-Flashable Software Bootloader*.)
2. Select a name for the secure boot signing key. This option will appear after secure boot is enabled. The file can be anywhere on your system. A relative path will be evaluated from the project directory. The file does not need to exist yet.
3. Set other menuconfig options (as desired). Pay particular attention to the “Bootloader Config” options, as you can only flash the bootloader once. Then exit menuconfig and save your configuration
4. The first time you run `make`, if the signing key is not found then an error message will be printed with a command to generate a signing key via `esptool.py generate_signing_key`.

**IMPORTANT** A signing key generated this way will use the best random number source available to the OS and its Python installation (`/dev/urandom` on OSX/Linux and `CryptGenRandom()` on Windows). If this random number source is weak, then the private key will be weak.

**IMPORTANT** For production environments, we recommend generating the keypair using `openssl` or another industry standard encryption program. See *Generating Secure Boot Signing Key* for more details.

5. Run `make bootloader` to build a secure boot enabled bootloader. The output of `make` will include a prompt for a flashing command, using `esptool.py write_flash`.
6. When you’re ready to flash the bootloader, run the specified command (you have to enter it yourself, this step is not performed by `make`) and then wait for flashing to complete. **Remember this is a one time flash, you can’t change the bootloader after this!**

7. Run `make flash` to build and flash the partition table and the just-built app image. The app image will be signed using the signing key you generated in step 4.

*NOTE:* `make flash` doesn't flash the bootloader if secure boot is enabled.

8. Reset the ESP32 and it will boot the software bootloader you flashed. The software bootloader will enable secure boot on the chip, and then it verifies the app image signature and boots the app. You should watch the serial console output from the ESP32 to verify that secure boot is enabled and no errors have occurred due to the build configuration.

**NOTE** Secure boot won't be enabled until after a valid partition table and app image have been flashed. This is to prevent accidents before the system is fully configured.

9. On subsequent boots, the secure boot hardware will verify the software bootloader has not changed (using the secure bootloader key) and then the software bootloader will verify the signed partition table and app image (using the public key portion of the secure boot signing key).

### 4.10.5 Re-Flashable Software Bootloader

Configuration "Secure Boot: One-Time Flash" is the recommended configuration for production devices. In this mode, each device gets a unique key that is never stored outside the device.

However, an alternative mode "Secure Boot: Reflashable" is also available. This mode allows you to supply a 256-bit key file that is used for the secure bootloader key. As you have the key file, you can generate new bootloader images and secure boot digests for them.

In the `esp-idf` build process, this 256-bit key file is derived from the app signing key generated during the `generate_signing_key` step above. The private key's SHA-256 digest is used as the 256-bit secure bootloader key. This is a convenience so you only need to generate/protect a single private key.

**NOTE:** Although it's possible, we strongly recommend not generating one secure boot key and flashing it to every device in a production environment. The "One-Time Flash" option is recommended for production environments.

To enable a reflashable bootloader:

1. In the `make menuconfig` step, select "Bootloader Config" -> "Secure Boot" -> "Reflashable".
2. Follow the steps shown above to choose a signing key file, and generate the key file.
3. Run `make bootloader`. A 256-bit key file will be created, derived from the private key that is used for signing. Two sets of flashing steps will be printed - the first set of steps includes an `espefuse.py burn_key` command which is used to write the bootloader key to efuse. (Flashing this key is a one-time-only process.) The second set of steps can be used to reflash the bootloader with a pre-calculated digest (generated during the build process).
4. Resume from *Step 6 of the one-time flashing process*, to flash the bootloader and enable secure boot. Watch the console log output closely to ensure there were no errors in the secure boot configuration.

### 4.10.6 Generating Secure Boot Signing Key

The build system will prompt you with a command to generate a new signing key via `espsecure.py generate_signing_key`. This uses the `python-ecdsa` library, which in turn uses Python's `os.urandom()` as a random number source.

The strength of the signing key is proportional to (a) the random number source of the system, and (b) the correctness of the algorithm used. For production devices, we recommend generating signing keys from a system with a quality entropy source, and using the best available EC key generation utilities.

For example, to generate a signing key using the `openssl` command line:

```
` openssl ecparam -name prime256v1 -genkey -noout -out my_secure_boot_signing_key.pem `
```

Remember that the strength of the secure boot system depends on keeping the signing key private.

### 4.10.7 Remote Signing of Images

For production builds, it can be good practice to use a remote signing server rather than have the signing key on the build machine (which is the default esp-idf secure boot configuration). The `espsecure.py` command line program can be used to sign app images & partition table data for secure boot, on a remote system.

To use remote signing, disable the option “Sign binaries during build”. The private signing key does not need to be present on the build system. However, the public (signature verification) key is required because it is compiled into the bootloader (and can be used to verify image signatures during OTA updates).

To extract the public key from the private key:

```
espsecure.py extract_public_key --keyfile PRIVATE_SIGNING_KEY PUBLIC_VERIFICATION_KEY
```

The path to the public signature verification key needs to be specified in the menuconfig under “Secure boot public signature verification key” in order to build the secure bootloader.

After the app image and partition table are built, the build system will print signing steps using `espsecure.py`:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY BINARY_FILE
```

The above command appends the image signature to the existing binary. You can use the `-output` argument to write the signed binary to a separate file:

```
espsecure.py sign_data --keyfile PRIVATE_SIGNING_KEY --output SIGNED_BINARY_FILE_↵  
↵BINARY_FILE
```

### 4.10.8 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.

### 4.10.9 Technical Details

The following sections contain low-level reference descriptions of various secure boot elements:

#### Secure Boot Hardware Support

The first stage of secure boot verification (checking the software bootloader) is done via hardware. The ESP32’s Secure Boot support hardware can perform three basic operations:

1. Generate a random sequence of bytes from a hardware random number generator.

2. Generate a digest from data (usually the bootloader image from flash) using a key stored in Efuse block 2. The key in Efuse can (& should) be read/write protected, which prevents software access. For full details of this algorithm see *Secure Bootloader Digest Algorithm*. The digest can only be read back by software if Efuse ABS\_DONE\_0 is *not* burned (ie still 0).
3. Generate a digest from data (usually the bootloader image from flash) using the same algorithm as step 2 and compare it to a pre-calculated digest supplied in a buffer (usually read from flash offset 0x0). The hardware returns a true/false comparison without making the digest available to software. This function is available even when Efuse ABS\_DONE\_0 is burned.

## Secure Bootloader Digest Algorithm

Starting with an “image” of binary data as input, this algorithm generates a digest as output. The digest is sometimes referred to as an “abstract” in hardware documentation.

For a Python version of this algorithm, see the `espsecure.py` tool in the `components/esptool_py` directory (specifically, the `digest_secure_bootloader` command).

Items marked with (^) are to fulfill hardware restrictions, as opposed to cryptographic restrictions.

1. Prefix the image with a 128 byte randomly generated IV.
2. If the image length is not modulo 128, pad the image to a 128 byte boundary with 0xFF. (^)
3. For each 16 byte plaintext block of the input image:
  - Reverse the byte order of the plaintext input block (^) -
  - Apply AES256 in ECB mode to the plaintext block. -
  - Reverse the byte order of the ciphertext output block. (^)
  - Append to the overall ciphertext output.
4. Byte-swap each 4 byte word of the ciphertext (^)
5. Calculate SHA-512 of the ciphertext.

Output digest is 192 bytes of data: The 128 byte IV, followed by the 64 byte SHA-512 digest.

## Image Signing Algorithm

Deterministic ECDSA as specified by [RFC 6979](#).

- Curve is NIST256p (openssl calls this curve “prime256v1”, it is also sometimes called secp256r1).
- Hash function is SHA256.
- Key format used for storage is PEM.
  - In the bootloader, the public key (for signature verification) is flashed as 64 raw bytes.
- Image signature is 68 bytes - a 4 byte version word (currently zero), followed by a 64 bytes of signature data. These 68 bytes are appended to an app image or partition table data.

## Manual Commands

Secure boot is integrated into the `esp-idf` build system, so `make` will automatically sign an app image if secure boot is enabled. `make bootloader` will produce a bootloader digest if `menuconfig` is configured for it.

However, it is possible to use the `espsecure.py` tool to make standalone signatures and digests.

To sign a binary image:

```
espsecure.py sign_data --keyfile ./my_signing_key.pem --output ./image_signed.bin_
↪image-unsigned.bin
```

Keyfile is the PEM file containing an ECDSA private signing key.

To generate a bootloader digest:

```
espsecure.py digest_secure_bootloader --keyfile ./securebootkey.bin --output ./
↳bootloader-digest.bin build/bootloader/bootloader.bin
```

Keyfile is the 32 byte raw secure boot key for the device. To flash this digest onto the device:

```
esptool.py write_flash 0x0 bootloader-digest.bin
```

## 4.10.10 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.

## 4.11 ULP coprocessor programming

### 4.11.1 ULP coprocessor instruction set

This document provides details about the instructions used by ESP32 ULP coprocessor assembler.

ULP coprocessor has 4 16-bit general purpose registers, labeled R0, R1, R2, R3. It also has an 8-bit counter register (stage\_cnt) which can be used to implement loops. Stage count register is accessed using special instructions.

ULP coprocessor can access 8k bytes of RTC\_SLOW\_MEM memory region. Memory is addressed in 32-bit word units. It can also access peripheral registers in RTC\_CNTL, RTC\_IO, and SENS peripherals.

All instructions are 32-bit. Jump instructions, ALU instructions, peripheral register and memory access instructions are executed in 1 cycle. Instructions which work with peripherals (TSENS, ADC, I2C) take variable number of cycles, depending on peripheral operation.

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily. This is true both for register names and instruction names.

#### Note about addressing

ESP32 ULP coprocessor’s JUMP, ST, LD instructions which take register as an argument (jump address, store/load base address) expect the argument to be expressed in 32-bit words.

Consider the following example program:

```
entry:
    NOP
    NOP
    NOP
    NOP
loop:
    MOVE R1, loop
    JUMP R1
```

When this program is assembled and linked, address of label `loop` will be equal to 16 (expressed in bytes). However *JUMP* instruction expects the address stored in register to be expressed in 32-bit words. To account for this common

use case, assembler will convert the address of label *loop* from bytes to words, when generating `MOVE` instruction, so the code generated code will be equivalent to:

```
0000    NOP
0004    NOP
0008    NOP
000c    NOP
0010    MOVE R1, 4
0014    JUMP R1
```

The other case is when the argument of `MOVE` instruction is not a label but a constant. In this case assembler will use the value as is, without any conversion:

```
.set      val, 0x10
MOVE     R1, val
```

In this case, value loaded into `R1` will be `0x10`.

Similar considerations apply to `LD` and `ST` instructions. Consider the following code:

```
.global array
array: .long 0
       .long 0
       .long 0
       .long 0

MOVE R1, array
MOVE R2, 0x1234
ST R2, R1, 0    // write value of R2 into the first array element,
                // i.e. array[0]

ST R2, R1, 4    // write value of R2 into the second array element
                // (4 byte offset), i.e. array[1]

ADD R1, R1, 2    // this increments address by 2 words (8 bytes)
ST R2, R1, 0    // write value of R2 into the third array element,
                // i.e. array[2]
```

### Note about instruction execution time

ULP coprocessor is clocked from `RTC_FAST_CLK`, which is normally derived from the internal 8MHz oscillator. Applications which need to know exact ULP clock frequency can calibrate it against the main XTAL clock:

```
#include "soc/rtc.h"

// calibrate 8M/256 clock against XTAL, get 8M/256 clock period
uint32_t rtc_8md256_period = rtc_clk_cal(RTC_CAL_8MD256, 100);
uint32_t rtc_fast_freq_hz = 1000000ULL * (1 << RTC_CLK_CAL_FRACT) * 256 / rtc_8md256_
↪period;
```

ULP coprocessor needs certain number of clock cycles to fetch each instruction, plus certain number of cycles to execute it, depending on the instruction. See description of each instruction below for details on the execution time.

Instruction fetch time is:

- 2 clock cycles — for instructions following ALU and branch instructions.
- 4 clock cycles — in other cases.

Note that when accessing RTC memories and RTC registers, ULP coprocessor has lower priority than the main CPUs. This means that ULP coprocessor execution may be suspended while the main CPUs access same memory region as the ULP.

## NOP - no operation

**Syntax** NOP

**Operands** None

**Cycles** 2 cycle to execute, 4 cycles to fetch next instruction

**Description** No operation is performed. Only the PC is incremented.

**Example:**

```
1:    NOP
```

## ADD - Add to register

**Syntax** ADD *Rdst*, *Rsrc1*, *Rsrc2*

ADD *Rdst*, *Rsrc1*, *imm*

**Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction adds source register to another source register or to a 16-bit signed value and stores result to the destination register.

**Examples:**

```
1:    ADD R1, R2, R3        //R1 = R2 + R3
2:    Add R1, R2, 0x1234   //R1 = R2 + 0x1234
3:    .set value1, 0x03    //constant value1=0x03
    Add R1, R2, value1    //R1 = R2 + value1
4:    .global label       //declaration of variable label
    Add R1, R2, label     //R1 = R2 + label
    ...
    label: nop            //definition of variable label
```

## SUB - Subtract from register

**Syntax** SUB *Rdst*, *Rsrc1*, *Rsrc2*

SUB *Rdst*, *Rsrc1*, *imm*

**Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction subtracts the source register from another source register or subtracts 16-bit signed value from a source register, and stores result to the destination register.

**Examples:**

```

1:      SUB R1, R2, R3           //R1 = R2 - R3
2:      sub R1, R2, 0x1234      //R1 = R2 - 0x1234
3:      .set value1, 0x03       //constant value1=0x03
        SUB R1, R2, value1      //R1 = R2 - value1
4:      .global label           //declaration of variable label
        SUB R1, R2, label       //R1 = R2 - label
        ....
label:  nop                     //definition of variable label

```

**AND - Logical AND of two operands**

**Syntax** **AND** *Rdst*, *Rsrc1*, *Rsrc2*

**AND** *Rdst*, *Rsrc1*, *imm*

**Operands**

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction does logical AND of a source register and another source register or 16-bit signed value and stores result to the destination register.

**Examples:**

```

1:      AND R1, R2, R3           //R1 = R2 & R3
2:      AND R1, R2, 0x1234      //R1 = R2 & 0x1234
3:      .set value1, 0x03       //constant value1=0x03
        AND R1, R2, value1      //R1 = R2 & value1
4:      .global label           //declaration of variable label
        AND R1, R2, label       //R1 = R2 & label
        ...
label:  nop                     //definition of variable label

```

## OR - Logical OR of two operands

**Syntax** `OR Rdst, Rsrc1, Rsrc2`

`OR Rdst, Rsrc1, imm`

### Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction does logical OR of a source register and another source register or 16-bit signed value and stores result to the destination register.

### Examples:

```
1:      OR R1, R2, R3           //R1 = R2 \| R3
2:      OR R1, R2, 0x1234      //R1 = R2 \| 0x1234
3:      .set value1, 0x03      //constant value1=0x03
        OR R1, R2, value1      //R1 = R2 \| value1
4:      .global label         //declaration of variable label
        OR R1, R2, label       //R1 = R2 \| label
        ...
label:  nop                    //definition of variable label
```

## LSH - Logical Shift Left

**Syntax** `LSH Rdst, Rsrc1, Rsrc2`

`LSH Rdst, Rsrc1, imm`

### Operands

- *Rdst* - Register R[0..3]
- *Rsrc1* - Register R[0..3]
- *Rsrc2* - Register R[0..3]
- *Imm* - 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction does logical shift to left of source register to number of bits from another source register or 16-bit signed value and store result to the destination register.

### Examples:

```
1:      LSH R1, R2, R3         //R1 = R2 << R3
2:      LSH R1, R2, 0x03      //R1 = R2 << 0x03
3:      .set value1, 0x03      //constant value1=0x03
```

(continues on next page)

(continued from previous page)

```

        LSH R1, R2, value1          //R1 = R2 << value1
4:      .global label              //declaration of variable label
        LSH R1, R2, label          //R1 = R2 << label
        ...
label:  nop                        //definition of variable label

```

## RSH - Logical Shift Right

**Syntax** **RSH** *Rdst, Rsrc1, Rsrc2*

**RSH** *Rdst, Rsrc1, imm*

**Operands** *Rdst* - Register R[0..3] *Rsrc1* - Register R[0..3] *Rsrc2* - Register R[0..3] *Imm* - 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction does logical shift to right of source register to number of bits from another source register or 16-bit signed value and store result to the destination register.

**Examples:**

```

1:      RSH R1, R2, R3              //R1 = R2 >> R3
2:      RSH R1, R2, 0x03           //R1 = R2 >> 0x03
3:      .set value1, 0x03          //constant value1=0x03
        RSH R1, R2, value1        //R1 = R2 >> value1
4:      .global label              //declaration of variable label
        RSH R1, R2, label         //R1 = R2 >> label
label:  nop                        //definition of variable label

```

## MOVE – Move to register

**Syntax** **MOVE** *Rdst, Rsrc*

**MOVE** *Rdst, imm*

**Operands**

- *Rdst* – Register R[0..3]
- *Rsrc* – Register R[0..3]
- *Imm* – 16-bit signed value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction move to destination register value from source register or 16-bit signed value.

Note that when a label is used as an immediate, the address of the label will be converted from bytes to words. This is because LD, ST, and JUMP instructions expect the address register value to be expressed in words rather than bytes. To avoid using an extra instruction

**Examples:**

```

1:      MOVE      R1, R2          //R1 = R2 >> R3
2:      MOVE      R1, 0x03       //R1 = R2 >> 0x03
3:      .set      value1, 0x03   //constant value1=0x03
      MOVE      R1, value1      //R1 = value1
4:      .global   label         //declaration of label
      MOVE      R1, label       //R1 = address_of(label) / 4
      ...
label:  nop                    //definition of label

```

## ST – Store data to the memory

**Syntax** *ST Rsrc, Rdst, offset*

### Operands

- *Rsrc* – Register R[0..3], holds the 16-bit value to store
- *Rdst* – Register R[0..3], address of the destination, in 32-bit words
- *Offset* – 10-bit signed value, offset in bytes

**Cycles** 4 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction stores the 16-bit value of *Rsrc* to the lower half-word of memory with address *Rdst*+*offset*. The upper half-word is written with the current program counter (PC), expressed in words, shifted left by 5 bits:

```
Mem[Rdst + offset / 4]{31:0} = {PC[10:0], 5'b0, Rsrc[15:0]}
```

The application can use higher 16 bits to determine which instruction in the ULP program has written any particular word into memory.

### Examples:

```

1:      ST   R1, R2, 0x12       //MEM[R2+0x12] = R1
2:      .data                               //Data section definition
Addr1: .word 123                 // Define label Addr1 16 bit
      .set  offs, 0x00          // Define constant offs
      .text                               //Text section definition
      MOVE R1, 1                // R1 = 1
      MOVE R2, Addr1           // R2 = Addr1
      ST   R1, R2, offs        // MEM[R2 + 0] = R1
                                      // MEM[Addr1 + 0] will be 32'h600001

```

## LD – Load data from the memory

**Syntax** *LD Rdst, Rsrc, offset*

**Operands** *Rdst* – Register R[0..3], destination

*Rsrc* – Register R[0..3], holds address of destination, in 32-bit words

*Offset* – 10-bit signed value, offset in bytes

**Cycles** 4 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction loads lower 16-bit half-word from memory with address  $Rsrc+offset$  into the destination register  $Rdst$ :

```
Rdst[15:0] = Mem[Rsrc + offset / 4][15:0]
```

#### Examples:

```
1:      LD   R1, R2, 0x12           //R1 = MEM[R2+0x12]
2:      .data                               //Data section definition
  Addr1: .word 123                    // Define label Addr1 16 bit
        .set  offs, 0x00             // Define constant offs
        .text                          //Text section definition
        MOVE   R1, 1                  // R1 = 1
        MOVE   R2, Addr1              // R2 = Addr1 / 4 (address of label is_
↳converted into words)
        LD     R1, R2, offs           // R1 = MEM[R2 + 0]
                                           // R1 will be 123
```

## JUMP – Jump to an absolute address

**Syntax** **JUMP** *Rdst*

**JUMP** *ImmAddr*

**JUMP** *Rdst, Condition*

**JUMP** *ImmAddr, Condition*

#### Operands

- *Rdst* – Register  $R[0..3]$  containing address to jump to (expressed in 32-bit words)
- *ImmAddr* – 13 bits address (expressed in bytes), aligned to 4 bytes
- **Condition:**
  - EQ – jump if last ALU operation result was zero
  - OV – jump if last ALU has set overflow flag

**Cycles** 2 cycles to execute, 2 cycles to fetch next instruction

**Description** The instruction makes jump to the specified address. Jump can be either unconditional or based on an ALU flag.

#### Examples:

```
1:      JUMP   R1                    // Jump to address in R1 (address in R1 is in 32-
↳bit words)
2:      JUMP   0x120, EQ             // Jump to address 0x120 (in bytes) if ALU result_
↳is zero
3:      JUMP   label                 // Jump to label
        ...
  label: nop                        // Definition of label
4:      .global label              // Declaration of global label
        MOVE   R1, label            // R1 = label (value loaded into R1 is in words)
```

(continues on next page)

(continued from previous page)

```

        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:**
  - *GE* (greater or equal) – jump if value in R0  $\geq$  threshold
  - *LT* (less than) – jump if value in R0  $<$  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 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  $\leq$  threshold
  - *GT* (greater than) – jump if value in stage\_cnt  $>$  threshold
  - *GE* (greater or equal) – jump if value in stage\_cnt  $\geq$  threshold

**Cycles** Conditions *LE*, *LT*, *GE*: 2 cycles to execute, 2 cycles to fetch next instruction

Conditions *EQ*, *GT* are implemented in the assembler using two **JUMPS** instructions:

```

// JUMPS target, threshold, EQ is implemented as:

    JUMPS next, threshold, LT
    JUMPS target, threshold, LE
next:

// JUMPS target, threshold, GT is implemented as:

    JUMPS next, threshold, LE
    JUMPS target, threshold, GE
next:

```

Therefore the execution time will depend on the branches taken: either 2 cycles to execute + 2 cycles to fetch, or 4 cycles to execute + 4 cycles to fetch.

**Description** The instruction makes a jump to a relative address if condition is true. Condition is the result of comparison of count register value and threshold value.

#### Examples:

```

1:pos:    JUMPS    16, 20, EQ    // Jump to (position + 16 bytes) if stage_cnt == 20

2:        // Up counting loop using stage count register
          STAGE_RST              // set stage_cnt to 0
label:    STAGE_INC 1            // stage_cnt++
          NOP                    // do something
          JUMPS    label, 16, LT // jump to label if stage_cnt < 16

```

### STAGE\_RST – Reset stage count register

#### Syntax STAGE\_RST

**Operands** No operands

**Description** The instruction sets the stage count register to 0

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

#### Examples:

```

1:        STAGE_RST    // Reset stage count register

```

### STAGE\_INC – Increment stage count register

#### Syntax STAGE\_INC *Value*

#### Operands

- *Value* – 8 bits value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction increments stage count register by given value.

#### Examples:

```

1:      STAGE_INC      10          // stage_cnt += 10

2:      // Up counting loop example:
      STAGE_RST          // set stage_cnt to 0
label:  STAGE_INC      1          // stage_cnt++
      NOP                // do something
      JUMPS            label, 16, LT // jump to label if stage_cnt < 16
    
```

## STAGE\_DEC – Decrement stage count register

**Syntax** STAGE\_DEC *Value*

### Operands

- *Value* – 8 bits value

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction decrements stage count register by given value.

### Examples:

```

1:      STAGE_DEC      10          // stage_cnt -= 10;

2:      // Down counting loop exaple
      STAGE_RST          // set stage_cnt to 0
      STAGE_INC      16          // increment stage_cnt to 16
label:  STAGE_DEC      1          // stage_cnt--;
      NOP                // do something
      JUMPS            label, 0, GT // jump to label if stage_cnt > 0
    
```

## HALT – End the program

**Syntax** HALT

**Operands** No operands

**Cycles** 2 cycles to execute

**Description** The instruction halts the ULP coprocessor and restarts ULP wakeup timer, if it is enabled.

### Examples:

```

1:      HALT           // Halt the coprocessor
    
```

## WAKE – Wake up the chip

**Syntax** WAKE

**Operands** No operands

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction sends an interrupt from ULP to RTC controller.

- If the SoC is in deep sleep mode, and ULP wakeup is enabled, this causes the SoC to wake up.
- If the SoC is not in deep sleep mode, and ULP interrupt bit (RTC\_CNTL\_ULP\_CP\_INT\_ENA) is set in RTC\_CNTL\_INT\_ENA\_REG register, RTC interrupt will be triggered.

Note that before using WAKE instruction, ULP program may needs to wait until RTC controller is ready to wake up the main CPU. This is indicated using RTC\_CNTL\_RDY\_FOR\_WAKEUP bit of RTC\_CNTL\_LOW\_POWER\_ST\_REG register. If WAKE instruction is executed while RTC\_CNTL\_RDY\_FOR\_WAKEUP is zero, it has no effect (wake up does not occur).

#### Examples:

```

1: is_rdy_for_wakeup:                // Read RTC_CNTL_RDY_FOR_WAKEUP bit
    READ_RTC_FIELD(RTC_CNTL_LOW_POWER_ST_REG, RTC_CNTL_RDY_FOR_WAKEUP)
    AND r0, r0, 1
    JUMP is_rdy_for_wakeup, eq      // Retry until the bit is set
    WAKE                            // Trigger wake up
    REG_WR 0x006, 24, 24, 0        // Stop ULP timer (clear RTC_CNTL_ULP_CP_SLP_
↳TIMER_EN)
    HALT                            // Stop the ULP program
    // After these instructions, SoC will wake up,
    // and ULP will not run again until started by the main program.

```

### SLEEP – set ULP wakeup timer period

**Syntax** SLEEP *sleep\_reg*

#### Operands

- *sleep\_reg* – 0..4, selects one of SENS\_ULP\_CP\_SLEEP\_CYCx\_REG registers.

**Cycles** 2 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction selects which of the SENS\_ULP\_CP\_SLEEP\_CYCx\_REG (x = 0..4) register values is to be used by the ULP wakeup timer as wakeup period. By default, the value from SENS\_ULP\_CP\_SLEEP\_CYC0\_REG is used.

#### Examples:

```

1:      SLEEP    1          // Use period set in SENS_ULP_CP_SLEEP_CYC1_REG

2:      .set sleep_reg, 4  // Set constant
        SLEEP  sleep_reg  // Use period set in SENS_ULP_CP_SLEEP_CYC4_REG

```

### WAIT – wait some number of cycles

**Syntax** WAIT *Cycles*

#### Operands

- *Cycles* – number of cycles for wait

**Cycles** 2 + *Cycles* cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction delays for given number of cycles.

#### Examples:

```

1:      WAIT     10         // Do nothing for 10 cycles

2:      .set  wait_cnt, 10  // Set a constant
        WAIT  wait_cnt     // wait for 10 cycles

```

## TSENS – do measurement with temperature sensor

### Syntax

- **TSENS** *Rdst, Wait\_Delay*

### Operands

- *Rdst* – Destination Register R[0..3], result will be stored to this register
- *Wait\_Delay* – number of cycles used to perform the measurement

**Cycles**  $2 + \textit{Wait\_Delay} + 3 * \text{TSENS\_CLK}$  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

**Cycles**  $23 + \max(1, \text{SAR\_AMP\_WAIT1}) + \max(1, \text{SAR\_AMP\_WAIT2}) + \max(1, \text{SAR\_AMP\_WAIT3}) + \text{SARx\_SAMPLE\_CYCLE} + \text{SARx\_SAMPLE\_BIT}$  cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction makes measurements from ADC.

### Examples:

```
1:      ADC        R1, 0, 1      // Measure value using ADC1 pad 2 and store result_
↳into R1
```

## I2C\_RD - read single byte from I2C slave

### Syntax

- **I2C\_RD** *Sub\_addr, High, Low, Slave\_sel*

### Operands

- *Sub\_addr* – Address within the I2C slave to read.
- *High, Low* — Define range of bits to read. Bits outside of [High, Low] range are masked.
- *Slave\_sel* - Index of I2C slave address to use.

**Cycles** Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

**Description** I2C\_RD instruction reads one byte from I2C slave with index *Slave\_sel*. Slave address (in 7-bit format) has to be set in advance into *SENS\_I2C\_SLAVE\_ADDRx* register field, where *x* == *Slave\_sel*. 8 bits of read result is stored into *R0* register.

**Examples:**

```
1:      I2C_RD      0x10, 7, 0, 0      // Read byte from sub-address 0x10 of slave_
↳with address set in SENS_I2C_SLAVE_ADDR0
```

## I2C\_WR - write single byte to I2C slave

**Syntax**

- **I2C\_WR** *Sub\_addr, Value, High, Low, Slave\_sel*

**Operands**

- *Sub\_addr* – Address within the I2C slave to write.
- *Value* – 8-bit value to be written.
- *High, Low* — Define range of bits to write. Bits outside of [High, Low] range are masked.
- *Slave\_sel* - Index of I2C slave address to use.

**Cycles** Execution time mostly depends on I2C communication time. 4 cycles to fetch next instruction.

**Description** I2C\_WR instruction writes one byte to I2C slave with index *Slave\_sel*. Slave address (in 7-bit format) has to be set in advance into *SENS\_I2C\_SLAVE\_ADDRx* register field, where *x* == *Slave\_sel*.

**Examples:**

```
1:      I2C_WR      0x20, 0x33, 7, 0, 1      // Write byte 0x33 to sub-address 0x20_
↳of slave with address set in SENS_I2C_SLAVE_ADDR1.
```

## REG\_RD – read from peripheral register

**Syntax** **REG\_RD** *Addr, High, Low*

**Operands**

- *Addr* – register address, in 32-bit words
- *High* – High part of R0
- *Low* – Low part of R0

**Cycles** 4 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction reads up to 16 bits from a peripheral register into a general purpose register:  $R0 = \text{REG}[\text{Addr}][\text{High}:\text{Low}]$ .

This instruction can access registers in RTC\_CNTL, RTC\_IO, and SENS peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

$$\text{addr\_ulp} = (\text{addr\_dport} - \text{DR\_REG\_RTCCNTL\_BASE}) / 4$$

**Examples:**

```
1:      REG_RD      0x120, 2, 0      // load 4 bits: R0 = {12'b0, REG[0x120][7:4]}
```

## REG\_WR – write to peripheral register

**Syntax** REG\_WR *Addr, High, Low, Data*

### Operands

- *Addr* – register address, in 32-bit words.
- *High* – High part of R0
- *Low* – Low part of R0
- *Data* – value to write, 8 bits

**Cycles** 8 cycles to execute, 4 cycles to fetch next instruction

**Description** The instruction writes up to 8 bits from a general purpose register into a peripheral register.  
REG[Addr][High:Low] = data

This instruction can access registers in RTC\_CNTL, RTC\_IO, and SENS peripherals. Address of the the register, as seen from the ULP, can be calculated from the address of the same register on the DPORT bus as follows:

```
addr_ulp = (addr_dport - DR_REG_RTC_CNTL_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`, and `soc/sens_reg.h`.

**READ\_RTC\_REG(*rtc\_reg, low\_bit, bit\_width*)** Read up to 16 bits from `rtc_reg[low_bit + bit_width - 1 : low_bit]` into R0. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Read 16 lower bits of RTC_CNTL_TIME0_REG into R0 */
READ_RTC_REG(RTC_CNTL_TIME0_REG, 0, 16)
```

**READ\_RTC\_FIELD(*rtc\_reg, field*)** Read from a field in `rtc_reg` into R0, up to 16 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/sens_reg.h"

/* Read 8-bit SENS_TSENS_OUT field of SENS_SAR_SLAVE_ADDR3_REG into R0 */
READ_RTC_FIELD(SENS_SAR_SLAVE_ADDR3_REG, SENS_TSENS_OUT)
```

**WRITE\_RTC\_REG(*rtc\_reg, low\_bit, bit\_width, value*)** Write immediate value into `rtc_reg[low_bit + bit_width - 1 : low_bit]`, `bit_width <= 8`. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_io_reg.h"

/* Set BIT(2) of RTC_GPIO_OUT_DATA_W1TS field in RTC_GPIO_OUT_W1TS_REG */
WRITE_RTC_REG(RTC_GPIO_OUT_W1TS_REG, RTC_GPIO_OUT_DATA_W1TS_S + 2, 1, 1)
```

**WRITE\_RTC\_FIELD(rtc\_reg, field, value)** Write immediate value into a field in rtc\_reg, up to 8 bits. For example:

```
#include "soc/soc_ulp.h"
#include "soc/rtc_cntl_reg.h"

/* Set RTC_CNTL_ULP_CP_SLP_TIMER_EN field of RTC_CNTL_STATE0_REG to 0 */
WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)
```

### 4.11.2 Programming ULP coprocessor using C macros

In addition to the existing binutils port for the ESP32 ULP coprocessor, it is possible to generate programs for the ULP by embedding assembly-like macros into an ESP32 application. Here is an example how this can be done:

```
const ulp_insn_t program[] = {
    I_MOVI(R3, 16),           // R3 <- 16
    I_LD(R0, R3, 0),         // R0 <- RTC_SLOW_MEM[R3 + 0]
    I_LD(R1, R3, 1),         // R1 <- RTC_SLOW_MEM[R3 + 1]
    I_ADDR(R2, R0, R1),      // R2 <- R0 + R1
    I_ST(R2, R3, 2),         // R2 -> RTC_SLOW_MEM[R2 + 2]
    I_HALT()
};
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);
```

The program array is an array of `ulp_insn_t`, i.e. ULP coprocessor instructions. Each `I_XXX` preprocessor define translates into a single 32-bit instruction. Arguments of these preprocessor defines can be register numbers (R0 -- R3) and literal constants. See *ULP coprocessor instruction defines* section for descriptions of instructions and arguments they take.

**Note:** Because some of the instruction macros expand to inline function calls, defining such array in global scope will cause the compiler to produce an “initializer element is not constant” error. To fix this error, move the definition of instructions array into local scope.

Load and store instructions use addresses expressed in 32-bit words. Address 0 corresponds to the first word of `RTC_SLOW_MEM` (which is address 0x50000000 as seen by the main CPUs).

To generate branch instructions, special `M_` preprocessor defines are used. `M_LABEL` define can be used to define a branch target. Label identifier is a 16-bit integer. `M_Bxxx` defines can be used to generate branch instructions with target set to a particular label.

Implementation note: these `M_` preprocessor defines will be translated into two `ulp_insn_t` values: one is a token value which contains label number, and the other is the actual instruction. `ulp_process_macros_and_load` function resolves the label number to the address, modifies the branch instruction to use the correct address, and removes the the extra `ulp_insn_t` token which contains the label number.

Here is an example of using labels and branches:

```

const ulp_insn_t program[] = {
    I_MOVI(R0, 34),           // R0 <- 34
    M_LABEL(1),              // label_1
    I_MOVI(R1, 32),           // R1 <- 32
    I_LD(R1, R1, 0),         // R1 <- RTC_SLOW_MEM[R1]
    I_MOVI(R2, 33),           // R2 <- 33
    I_LD(R2, R2, 0),         // R2 <- RTC_SLOW_MEM[R2]
    I_SUBR(R3, R1, R2),       // R3 <- R1 - R2
    I_ST(R3, R0, 0),         // R3 -> RTC_SLOW_MEM[R0 + 0]
    I_ADDI(R0, R0, 1),       // R0++
    M_BL(1, 64),             // if (R0 < 64) goto label_1
    I_HALT(),
};
RTC_SLOW_MEM[32] = 42;
RTC_SLOW_MEM[33] = 18;
size_t load_addr = 0;
size_t size = sizeof(program)/sizeof(ulp_insn_t);
ulp_process_macros_and_load(load_addr, program, &size);
ulp_run(load_addr);

```

## Functions

`esp_err_t ulp_process_macros_and_load` (uint32\_t *load\_addr*, const ulp\_insn\_t \**program*, size\_t *psize*)

Resolve all macro references in a program and load it into RTC memory.

### Return

- ESP\_OK on success
- ESP\_ERR\_NO\_MEM if auxiliary temporary structure can not be allocated
- one of ESP\_ERR\_ULP\_xxx if program is not valid or can not be loaded

### Parameters

- *load\_addr*: address where the program should be loaded, expressed in 32-bit words
- *program*: ulp\_insn\_t array with the program
- *psize*: size of the program, expressed in 32-bit words

`esp_err_t ulp_run` (uint32\_t *entry\_point*)

Run the program loaded into RTC memory.

**Return** ESP\_OK on success

### Parameters

- *entry\_point*: entry point, expressed in 32-bit words

## Error codes

### ESP\_ERR\_ULP\_BASE

Offset for ULP-related error codes

### ESP\_ERR\_ULP\_SIZE\_TOO\_BIG

Program doesn't fit into RTC memory reserved for the ULP

**ESP\_ERR\_ULP\_INVALID\_LOAD\_ADDR**

Load address is outside of RTC memory reserved for the ULP

**ESP\_ERR\_ULP\_DUPLICATE\_LABEL**

More than one label with the same number was defined

**ESP\_ERR\_ULP\_UNDEFINED\_LABEL**

Branch instructions references an undefined label

**ESP\_ERR\_ULP\_BRANCH\_OUT\_OF\_RANGE**

Branch target is out of range of B instruction (try replacing with BX)

## ULP coprocessor registers

ULP co-processor has 4 16-bit general purpose registers. All registers have same functionality, with one exception. R0 register is used by some of the compare-and-branch instructions as a source register.

These definitions can be used for all instructions which require a register.

**R0**

general purpose register 0

**R1**

general purpose register 1

**R2**

general purpose register 2

**R3**

general purpose register 3

## ULP coprocessor instruction defines

**I\_DELAY** (cycles\_)

Delay (nop) for a given number of cycles

**I\_HALT** ()

Halt the coprocessor.

This instruction halts the coprocessor, but keeps ULP timer active. As such, ULP program will be restarted again by timer. To stop the program and prevent the timer from restarting the program, use I\_END(0) instruction.

**I\_END** ()

Stop ULP program timer.

This is a convenience macro which disables the ULP program timer. Once this instruction is used, ULP program will not be restarted anymore until ulp\_run function is called.

ULP program will continue running after this instruction. To stop the currently running program, use I\_HALT().

**I\_ST** (reg\_val, reg\_addr, offset\_)

Store value from register reg\_val into RTC memory.

The value is written to an offset calculated by adding value of reg\_addr register and offset\_ field (this offset is expressed in 32-bit words). 32 bits written to RTC memory are built as follows:

- bits [31:21] hold the PC of current instruction, expressed in 32-bit words
- bits [20:16] = 5'b1
- bits [15:0] are assigned the contents of reg\_val

$RTC\_SLOW\_MEM[addr + offset\_] = \{ 5'b0, insn\_PC[10:0], val[15:0] \}$

**I\_LD** (reg\_dest, reg\_addr, offset\_)

Load value from RTC memory into reg\_dest register.

Loads 16 LSBs from RTC memory word given by the sum of value in reg\_addr and value of offset\_.

**I\_WR\_REG** (reg, low\_bit, high\_bit, val)

Write literal value to a peripheral register

$reg[high\_bit : low\_bit] = val$  This instruction can access RTC\_CNTL\_, RTC\_IO\_, and SENS\_ peripheral registers.

**I\_RD\_REG** (reg, low\_bit, high\_bit)

Read from peripheral register into R0

$R0 = reg[high\_bit : low\_bit]$  This instruction can access RTC\_CNTL\_, RTC\_IO\_, and SENS\_ peripheral registers.

**I\_BL** (pc\_offset, imm\_value)

Branch relative if R0 less than immediate value.

pc\_offset is expressed in words, and can be from -127 to 127 imm\_value is a 16-bit value to compare R0 against

**I\_BGE** (pc\_offset, imm\_value)

Branch relative if R0 greater or equal than immediate value.

pc\_offset is expressed in words, and can be from -127 to 127 imm\_value is a 16-bit value to compare R0 against

**I\_BXR** (reg\_pc)

Unconditional branch to absolute PC, address in register.

reg\_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

**I\_BXI** (imm\_pc)

Unconditional branch to absolute PC, immediate address.

Address imm\_pc is expressed in 32-bit words.

**I\_BXZR** (reg\_pc)

Branch to absolute PC if ALU result is zero, address in register.

reg\_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

**I\_BXZI** (imm\_pc)

Branch to absolute PC if ALU result is zero, immediate address.

Address imm\_pc is expressed in 32-bit words.

**I\_BXFR** (reg\_pc)

Branch to absolute PC if ALU overflow, address in register

reg\_pc is the register which contains address to jump to. Address is expressed in 32-bit words.

**I\_BXFI** (imm\_pc)

Branch to absolute PC if ALU overflow, immediate address

Address imm\_pc is expressed in 32-bit words.

**I\_ADDR** (reg\_dest, reg\_src1, reg\_src2)

Addition:  $dest = src1 + src2$

**I\_SUBR** (reg\_dest, reg\_src1, reg\_src2)

Subtraction:  $dest = src1 - src2$

**I\_ANDR** (reg\_dest, reg\_src1, reg\_src2)

Logical AND: dest = src1 & src2

**I\_ORR** (reg\_dest, reg\_src1, reg\_src2)

Logical OR: dest = src1 | src2

**I\_MOVR** (reg\_dest, reg\_src)

Copy: dest = src

**I\_LSHR** (reg\_dest, reg\_src, reg\_shift)

Logical shift left: dest = src << shift

**I\_RSHR** (reg\_dest, reg\_src, reg\_shift)

Logical shift right: dest = src >> shift

**I\_ADDI** (reg\_dest, reg\_src, imm\_)

Add register and an immediate value: dest = 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

**M\_LABEL** (label\_num)

Define a label with number label\_num.

This is a macro which doesn't generate a real instruction. The token generated by this macro is removed by ulp\_process\_macros\_and\_load function. Label defined using this macro can be used in branch macros defined below.

**M\_BL** (label\_num, imm\_value)

Macro: branch to label label\_num if R0 is less than immediate value.

This macro generates two ulp\_insn\_t values separated by a comma, and should be used when defining contents of ulp\_insn\_t arrays. First value is not a real instruction; it is a token which is removed by ulp\_process\_macros\_and\_load function.

**M\_BGE** (label\_num, imm\_value)

Macro: branch to label label\_num if R0 is greater or equal than immediate value

This macro generates two ulp\_insn\_t values separated by a comma, and should be used when defining contents of ulp\_insn\_t arrays. First value is not a real instruction; it is a token which is removed by ulp\_process\_macros\_and\_load function.

**M\_BX** (label\_num)

Macro: unconditional branch to label

This macro generates two ulp\_insn\_t values separated by a comma, and should be used when defining contents of ulp\_insn\_t arrays. First value is not a real instruction; it is a token which is removed by ulp\_process\_macros\_and\_load function.

**M\_BXZ** (label\_num)

Macro: branch to label if ALU result is zero

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

**M\_BXF** (label\_num)

Macro: branch to label if ALU overflow

This macro generates two `ulp_insn_t` values separated by a comma, and should be used when defining contents of `ulp_insn_t` arrays. First value is not a real instruction; it is a token which is removed by `ulp_process_macros_and_load` function.

## Defines

**RTC\_SLOW\_MEM**

RTC slow memory, 8k size

ULP (Ultra Low Power) coprocessor is a simple FSM which is designed to perform measurements using ADC, temperature sensor, and external I2C sensors, while main processors are in deep sleep mode. ULP coprocessor can access `RTC_SLOW_MEM` memory region, and registers in `RTC_CNTL`, `RTC_IO`, and `SARADC` peripherals. ULP coprocessor uses fixed-width 32-bit instructions, 32-bit memory addressing, and has 4 general purpose 16-bit registers.

### 4.11.3 Installing the toolchain

ULP coprocessor code is written in assembly and compiled using the `binutils-esp32ulp` toolchain.

1. Download pre-built binaries of the latest toolchain release from: <https://github.com/espressif/binutils-esp32ulp/releases>.
2. Extract the toolchain into a directory, and add the path to the `bin/` directory of the toolchain to the `PATH` environment variable.

### 4.11.4 Compiling ULP code

To compile ULP code as part of a component, the following steps must be taken:

1. ULP code, written in assembly, must be added to one or more files with `.S` extension. These files must be placed into a separate directory inside component directory, for instance `ulp/`.
2. Modify the component makefile, adding the following:

```
ULP_APP_NAME ?= ulp_$(COMPONENT_NAME)
ULP_S_SOURCES = $(COMPONENT_PATH)/ulp/ulp_source_file.S
ULP_EXP_DEP_OBJECTS := main.o
include $(IDF_PATH)/components/ulp/component_ulp_common.mk
```

Here is each line explained:

**ULP\_APP\_NAME** Name of the generated ULP application, without an extension. This name is used for build products of the ULP application: ELF file, map file, binary file, generated header file, and generated linker export file.

**ULP\_S\_SOURCES** List of assembly files to be passed to the ULP assembler. These must be absolute paths, i.e. start with `$(COMPONENT_PATH)`. Consider using `$(addprefix)` function if more than one file needs to be listed. Paths are relative to component build directory, so prefixing them is not necessary.

**ULP\_EXP\_DEP\_OBJECTS** List of object files names within the component which include the generated header file. This list is needed to build the dependencies correctly and ensure that the generated header file is created before any of these files are compiled. See section below explaining the concept of generated header files for ULP applications.

**include \$(IDF\_PATH)/components/ulp/component\_ulp\_common.mk** Includes common definitions of ULP build steps. Defines build targets for ULP object files, ELF file, binary file, etc.

3. Build the application as usual (e.g. *make app*)

Inside, the build system will take the following steps to build ULP program:

1. **Run each assembly file (foo.S) through C preprocessor.** This step generates the preprocessed assembly files (foo.ulp.pS) in the component build directory. This step also generates dependency files (foo.ulp.d).
2. **Run preprocessed assembly sources through assembler.** This produces objects (foo.ulp.o) and listing (foo.ulp.lst) files. Listing files are generated for debugging purposes and are not used at later stages of build process.
3. **Run linker script template through C preprocessor.** The template is located in components/ulp/ld directory.
4. **Link object files into an output ELF file (ulp\_app\_name.elf).** Map file (ulp\_app\_name.map) generated at this stage may be useful for debugging purposes.
5. **Dump contents of the ELF file into binary (ulp\_app\_name.bin)** for embedding into the application.
6. **Generate list of global symbols (ulp\_app\_name.sym)** in the ELF file using esp32ulp-elf-nm.
7. **Create LD export script and header file (ulp\_app\_name.ld and ulp\_app\_name.h)** containing the symbols from ulp\_app\_name.sym. This is done using esp32ulp\_mapgen.py utility.
8. **Add the generated binary to the list of binary files** to be emedded into the application.

#### 4.11.5 Accessing ULP program variables

Global symbols defined in the ULP program may be used inside the main program.

For example, ULP program may define a variable `measurement_count` which will define the number of ADC measurements the program needs to make before waking up the chip from deep sleep:

```

measurement_count:    .global measurement_count
                    .long 0

                    /* later, use measurement_count */
                    move r3, measurement_count
                    ld r3, r3, 0

```

Main program needs to initialize this variable before ULP program is started. Build system makes this possible by generating a `$(ULP_APP_NAME).h` and `$(ULP_APP_NAME).ld` files which define global symbols present in the ULP program. This files include each global symbol defined in the ULP program, prefixed with `ulp_`.

The header file contains declaration of the symbol:

```
extern uint32_t ulp_measurement_count;
```

Note that all symbols (variables, arrays, functions) are declared as `uint32_t`. For functions and arrays, take address of the symbol and cast to the appropriate type.

The generated linker script file defines locations of symbols in `RTC_SLOW_MEM`:

```
PROVIDE ( ulp_measurement_count = 0x50000060 );
```

To access ULP program variables from the main program, include the generated header file and use variables as one normally would:

```
#include "ulp_app_name.h"

// later
void init_ulp_vars() {
    ulp_measurement_count = 64;
}
```

Note that ULP program can only use lower 16 bits of each 32-bit word in RTC memory, because the registers are 16-bit, and there is no instruction to load from high part of the word.

Likewise, ULP store instruction writes register value into the lower 16 bit part of the 32-bit word. Upper 16 bits are written with a value which depends on the address of the store instruction, so when reading variables written by the ULP, main application needs to mask upper 16 bits, e.g.:

```
printf("Last measurement value: %d\n", ulp_last_measurement & UINT16_MAX);
```

#### 4.11.6 Starting the ULP program

To run a ULP program, main application needs to load the ULP program into RTC memory using `ulp_load_binary` function, and then start it using `ulp_run` function.

Note that “Enable Ultra Low Power (ULP) Coprocessor” option must be enabled in menuconfig in order to reserve memory for the ULP. “RTC slow memory reserved for coprocessor” option must be set to a value sufficient to store ULP code and data. If the application components contain multiple ULP programs, then the size of the RTC memory must be sufficient to hold the largest one.

Each ULP program is embedded into the ESP-IDF application as a binary blob. Application can reference this blob and load it in the following way (suppose `ULP_APP_NAME` was defined to `ulp_app_name`:

```
extern const uint8_t bin_start[] asm("_binary_ulp_app_name_bin_start");
extern const uint8_t bin_end[]   asm("_binary_ulp_app_name_bin_end");

void start_ulp_program() {
    ESP_ERROR_CHECK( ulp_load_binary(
        0 /* load address, set to 0 when using default linker scripts */,
        bin_start,
        (bin_end - bin_start) / sizeof(uint32_t) );
}
```

`esp_err_t ulp_load_binary` (`uint32_t load_addr`, `const uint8_t *program_binary`, `size_t program_size`)  
Load ULP program binary into RTC memory.

ULP program binary should have the following format (all values little-endian):

1. MAGIC, (value 0x00706c75, 4 bytes)
2. TEXT\_OFFSET, offset of .text section from binary start (2 bytes)
3. TEXT\_SIZE, size of .text section (2 bytes)
4. DATA\_SIZE, size of .data section (2 bytes)
5. BSS\_SIZE, size of .bss section (2 bytes)

6. (TEXT\_OFFSET - 12) bytes of arbitrary data (will not be loaded into RTC memory)
7. .text section
8. .data section

Linker script in components/ulp/ld/esp32.ulp.ld produces ELF files which correspond to this format. This linker script produces binaries with load\_addr == 0.

#### Return

- ESP\_OK on success
- ESP\_ERR\_INVALID\_ARG if load\_addr is out of range
- ESP\_ERR\_INVALID\_SIZE if program\_size doesn't match (TEXT\_OFFSET + TEXT\_SIZE + DATA\_SIZE)
- ESP\_ERR\_NOT\_SUPPORTED if the magic number is incorrect

#### Parameters

- load\_addr: address where the program should be loaded, expressed in 32-bit words
- program\_binary: pointer to program binary
- program\_size: size of the program binary

Once the program is loaded into RTC memory, application can start it, passing the address of the entry point to ulp\_run function:

```
ESP_ERROR_CHECK( ulp_run(&ulp_entry - RTC_SLOW_MEM) );
```

esp\_err\_t **ulp\_run** (uint32\_t *entry\_point*)

Run the program loaded into RTC memory.

**Return** ESP\_OK on success

#### Parameters

- entry\_point: entry point, expressed in 32-bit words

Declaration of the entry point symbol comes from the above mentioned generated header file, \$(ULP\_APP\_NAME).h. In assembly source of the ULP application, this symbol must be marked as .global:

```
.global entry
entry:
    /* code starts here */
```

### 4.11.7 ULP program flow

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 150kHz RC oscillator). The number of ticks is set using SENS\_ULP\_CP\_SLEEP\_CYCx\_REG registers (x = 0..4). When starting the ULP for the first time, SENS\_ULP\_CP\_SLEEP\_CYC0\_REG will be used to set the number of timer ticks. Later the ULP program can select another SENS\_ULP\_CP\_SLEEP\_CYCx\_REG register using sleep instruction.

The application can set ULP timer period values (SENS\_ULP\_CP\_SLEEP\_CYCx\_REG, x = 0..4) using ulp\_wakeup\_period\_set function.

`esp_err_t ulp_set_wakeup_period` (`size_t period_index`, `uint32_t period_us`)  
Set one of ULP wakeup period values.

ULP coprocessor starts running the program when the wakeup timer counts up to a given value (called period). There are 5 period values which can be programmed into `SENS_ULP_CP_SLEEP_CYCx_REG` registers,  $x = 0..4$ . By default, wakeup timer will use the period set into `SENS_ULP_CP_SLEEP_CYC0_REG`, i.e. period number 0. ULP program code can use `SLEEP` instruction to select which of the `SENS_ULP_CP_SLEEP_CYCx_REG` should be used for subsequent wakeups.

#### Return

- `ESP_OK` on success
- `ESP_ERR_INVALID_ARG` if `period_index` is out of range

#### Parameters

- `period_index`: wakeup period setting number (0 - 4)
- `period_us`: wakeup period, us

Once the timer counts the number of ticks set in the selected `SENS_ULP_CP_SLEEP_CYCx_REG` register, 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_STATE0_REG` register. This can be done both from ULP code and from the main program.

## 4.12 Unit Testing in ESP32

ESP-IDF comes with a unit test app based on Unity - unit test framework. Unit tests are integrated in the ESP-IDF repository and are placed in `test` subdirectory of each component respectively.

### 4.12.1 Adding unit tests

Unit tests are added in the `test` subdirectory of the respective component. Tests are added in C files, a single C file can include multiple test cases. Test files start with the word “test”.

The test file should include `unity.h` and the header for the C module to be tested.

Tests are added in a function in the C file as follows:

```
TEST_CASE("test name", "[module name]"
{
    // Add test here
})
```

First argument is a descriptive name for the test, second argument is an identifier in square brackets. Identifiers are used to group related test, or tests with specific properties.

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()`, run the tests cases, and then call `UNITY_END()`.

Each `test` subdirectory needs to include `component.mk` file with at least the following line of code:

```
COMPONENT_ADD_LDFLAGS = -Wl,--whole-archive -l$(COMPONENT_NAME) -Wl,--no-whole-archive
```

See <http://www.throwtheswitch.org/unity> for more information about writing tests in Unity.

### 4.12.2 Building unit test app

Follow the setup instructions in the top-level esp-idf README. Make sure that `IDF_PATH` environment variable is set to point to the path of esp-idf top-level directory.

Change into `tools/unit-test-app` directory to configure and build it:

- `make menuconfig` - configure unit test app.
- `make TESTS_ALL=1` - build unit test app with tests for each component having tests in the `test` subdirectory.
- `make TEST_COMPONENTS='xxx'` - build unit test app with tests for specific components.

When the build finishes, it will print instructions for flashing the chip. You can simply run `make flash` to flash all build output.

You can also run `make flash TESTS_ALL=1` or `make TEST_COMPONENTS='xxx'` to build and flash. Everything needed will be rebuilt automatically before flashing.

Use `menuconfig` to set the serial port for flashing.

### 4.12.3 Running unit tests

After flashing reset the ESP32 and it will boot the unit test app.

Unit test app prints a test menu with all available tests.

Test cases can be run by inputting one of the following:

- Test case name in quotation marks to run a single test case
- Test case index to run a single test case
- Module name in square brackets to run all test cases for a specific module
- An asterisk to run all test cases

## 4.13 Console

ESP-IDF provides `console` component, which includes building blocks needed to develop an interactive console over serial port. This component includes following facilities:

- Line editing, provided by `linenoise` library. This includes handling of backspace and arrow keys, scrolling through command history, command auto-completion, and argument hints.
- Splitting of command line into arguments.
- Argument parsing, provided by `argtable3` library. This library includes APIs useful for parsing GNU style command line arguments.
- Functions for registration and dispatching of commands.

These facilities can be used together or independently. For example, it is possible to use line editing and command registration features, but use `getopt` or custom code for argument parsing, instead of `argtable3`. Likewise, it is possible to use simpler means of command input (such as `fgets`) together with the rest of the means for command splitting and argument parsing.

### 4.13.1 Line editing

Line editing feature lets users compose commands by typing them, erasing symbols using ‘backspace’ key, navigating within the command using left/right keys, navigating to previously typed commands using up/down keys, and performing autocompletion using ‘tab’ key.

---

**Note:** This feature relies on ANSI escape sequence support in the terminal application. As such, serial monitors which display raw UART data can not be used together with the line editing library. If you see `[\6n` or similar escape sequence when running `get_started/console` example instead of a command prompt (`[esp32]>`), it means that the serial monitor does not support escape sequences. Programs which are known to work are GNU screen, minicom, and `idf_monitor.py` (which can be invoked using `make monitor` from project directory).

---

Here is an overview of functions provided by `linenoise` library.

#### Configuration

Linenoise library does not need explicit initialization. However, some configuration defaults may need to be changed before invoking the main line editing function.

**linenoiseClearScreen** Clear terminal screen using an escape sequence and position the cursor at the top left corner.

**linenoiseSetMultiLine** Switch between single line and multi line editing modes. In single line mode, if the length of the command exceeds the width of the terminal, the command text is scrolled within the line to show the end of the text. In this case the beginning of the text is hidden. Single line needs less data to be sent to refresh screen on each key press, so exhibits less glitching compared to the multi line mode. On the flip side, editing commands and copying command text from terminal in single line mode is harder. Default is single line mode.

#### Main loop

**linenoise** In most cases, console applications have some form of read/eval loop. `linenoise` is the single function which handles user’s key presses and returns completed line once ‘enter’ key is pressed. As such, it handles the ‘read’ part of the loop.

**linenoiseFree** This function must be called to release the command line buffer obtained from `linenoise` function.

#### Hints and completions

**linenoiseSetCompletionCallback** When user presses ‘tab’ key, `linenoise` library invokes completion callback. The callback should inspect the contents of the command typed so far and provide a list of possible completions using calls to `linenoiseAddCompletion` function. `linenoiseSetCompletionCallback` function should be called to register this completion callback, if completion feature is desired.

`console` component provides a ready made function to provide completions for registered commands, `esp_console_get_completion` (see below).

**linenoiseAddCompletion** Function to be called by completion callback to inform the library about possible completions of the currently typed command.

**linenoiseSetHintsCallback** Whenever user input changes, linenoise invokes hints callback. This callback can inspect the command line typed so far, and provide a string with hints (which can include list of command arguments, for example). The library then displays the hint text on the same line where editing happens, possibly with a different color.

**linenoiseSetFreeHintsCallback** If hint string returned by hints callback is dynamically allocated or needs to be otherwise recycled, the function which performs such cleanup should be registered via `linenoiseSetFreeHintsCallback`.

## History

**linenoiseHistorySetMaxLen** This function sets the number of most recently typed commands to be kept in memory. Users can navigate the history using up/down arrows.

**linenoiseHistoryAdd** Linenoise does not automatically add commands to history. Instead, applications need to call this function to add command strings to the history.

**linenoiseHistorySave** Function saves command history from RAM to a text file, for example on an SD card or on a filesystem in flash memory.

**linenoiseHistoryLoad** Counterpart to `linenoiseHistorySave`, loads history from a file.

**linenoiseHistoryFree** Releases memory used to store command history. Call this function when done working with linenoise library.

### 4.13.2 Splitting of command line into arguments

`console` component provides `esp_console_split_argv` function to split command line string into arguments. The function returns the number of arguments found (`argc`) and fills an array of pointers which can be passed as `argv` argument to any function which accepts arguments in `argc, argv` format.

The command line is split into arguments according to the following rules:

- Arguments are separated by spaces
- If spaces within arguments are required, they can be escaped using `\` (backslash) character.
- Other escape sequences which are recognized are `\\` (which produces literal backslash) and `\"`, which produces a double quote.
- Arguments can be quoted using double quotes. Quotes may appear only in the beginning and at the end of the argument. Quotes within the argument must be escaped as mentioned above. Quotes surrounding the argument are stripped by `esp_console_split_argv` function.

Examples:

- `abc def 1 20 .3 [ abc, def, 1, 20, .3 ]`
- `abc "123 456" def [ abc, 123 456, def ]`
- ``a\ b\\c\" [ a b\c" ]`

### 4.13.3 Argument parsing

For argument parsing, `console` component includes `argtable3` library. Please see [tutorial](#) for an introduction to `argtable3`. Github repository also includes [examples](#).

### 4.13.4 Command registration and dispatching

`console` component includes utility functions which handle registration of commands, matching commands typed by the user to registered ones, and calling these commands with the arguments given on the command line.

Application first initializes command registration module using a call to `esp_console_init`, and calls `esp_console_cmd_register` function to register command handlers.

For each command, application provides the following information (in the form of `esp_console_cmd_t` structure):

- Command name (string without spaces)
- Help text explaining what the command does
- Optional hint text listing the arguments of the command. If application uses `Argtable3` for argument parsing, hint text can be generated automatically by providing a pointer to `argtable` argument definitions structure instead.
- The command handler function.

A few other functions are provided by the command registration module:

**`esp_console_run`** This function takes the command line string, splits it into `argc/argv` argument list using `esp_console_split_argv`, looks up the command in the list of registered components, and if it is found, executes its handler.

**`esp_console_split_argv`** 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`.

### 4.13.5 Example

Example application illustrating usage of the `console` component is available in `examples/system/console` directory. This example shows how to initialize UART and VFS functions, set up `linenoise` library, read and handle commands from UART, and store command history in Flash. See `README.md` in the example directory for more details.

## 4.14 ESP32 ROM console

When an ESP32 is unable to boot from flash ROM (and the fuse disabling it hasn't been blown), it boots into a rom console. The console is based on TinyBasic, and statements entered should be in the form of BASIC statements. As is common in the BASIC language, without a preceding line number, commands entered are executed immediately; lines with a prefixed line number are stored as part of a program.

### 4.14.1 Full list of supported statements and functions

#### System

- `BYE` - *exits Basic, reboots ESP32, retries booting from flash*
- `END` - *stops execution from the program, also "STOP"*

- MEM - *displays memory usage statistics*
- NEW - *clears the current program*
- RUN - *executes the current program*

## IO, Documentation

- PEEK( address ) - *get a 32-bit value from a memory address*
- POKE - *write a 32-bit value to memory*
- USR(addr, arg1, ..) - *Execute a machine language function*
- PRINT expression - *print out the expression, also “?”*
- PHEX expression - *print expression as a hex number*
- REM stuff - *remark/comment, also “”*

## Expressions, Math

- A=V, LET A=V - *assign value to a variable*
- +, -, \*, / - *Math*
- <, <=, =, <>, !=, >=, > - *Comparisons*
- ABS( expression ) - *returns the absolute value of the expression*
- RSEED( v ) - *sets the random seed to v*
- RND( m ) - *returns a random number from 0 to m*
- A=1234 - \* Assign a decimal value\*
- A=&h1A2 - \* Assign a hex value\*
- A=&b1001 - *Assign a binary value*

## Control

- IF expression THEN statement - *perform statement if expression is true*
- FOR variable = start TO end - *start for block*
- FOR variable = start TO end STEP value - *start for block with step*
- NEXT - *end of for block*
- GOTO linenummer - *continue execution at this line number*
- GOSUB linenummer - *call a subroutine at this line number*
- RETURN - *return from a subroutine*
- DELAY - *Delay a given number of milliseconds*

## Pin IO

- IODIR - *Set a GPIO-pin as an output (1) or input (0)*
- IOSET - *Set a GPIO-pin, configured as output, to high (1) or low (0)*
- IOGET - *Get the value of a GPIO-pin*

### 4.14.2 Example programs

Here are a few example commands and programs to get you started...

#### Read UART\_DATE register of uart0

```
> PHEX PEEK(&h3FF40078)
15122500
```

#### Set GPIO2 using memory writes to GPIO\_OUT\_REG

Note: you can do this easier with the IOSET command

```
> POKE &h3FF44004,PEEK(&h3FF44004) OR &b100
```

#### Get value of GPIO0

```
> IODIR 0,0
> PRINT IOGET(0)
0
```

#### Blink LED

Hook up an LED between GPIO2 and ground. When running the program, the LED should blink 10 times.

```
10 IODIR 2,1
20 FOR A=1 TO 10
30 IOSET 2,1
40 DELAY 250
50 IOSET 2,0
60 DELAY 250
70 NEXT A
RUN
```

### 4.14.3 Credits

The ROM console is based on “TinyBasicPlus” by Mike Field and Scott Lawrence, which is based on “68000 Tiny-Basic” by Gordon Brandy

## 4.15 Wi-Fi Driver

### 4.15.1 Important Notes

- This document describes the implementation of only the **latest** IDF release. Backward compatibility with older versions of ESP-IDF is not guaranteed.
- This document describes the features which have already been implemented in the **latest** IDF release. For features that are now in developing/testing status, we also provide brief descriptions, while indicating the release versions in which these features will be eventually implemented.
- If you find anything wrong/ambiguous/hard to understand or inconsistent with the implementation, feel free to let us know about it on our IDF GitHub page.

### 4.15.2 ESP32 Wi-Fi Feature List

- Supports Station-only mode, SoftAP-only mode, Station/SoftAP-coexistence mode
- Supports IEEE-802.11B, IEEE-802.11G, IEEE802.11N and APIs to configure the protocol mode
- Supports WPA/WPA2/WPA2-Enterprise and WPS
- Supports AMPDU, HT40, QoS and other key features
- Supports Modem-sleep
- Supports an Espressif-specific protocol which, in turn, supports up to **1 km** of data traffic
- Up to 20 MBit/sec TCP throughput and 30 MBit/sec UDP throughput over the air
- Supports Sniffer
- Support set fast\_crypto algorithm and normal algorithm switch which used in wifi connect
- Support both fast scan and all channel scan feature

### 4.15.3 How To Write a Wi-Fi Application

#### Preparation

Generally, the most effective way to begin your own Wi-Fi application is to select an example which is similar to your own application, and port the useful part into your project. It is not a **MUST** but it is strongly recommended that you take some time to read this article first, especially if you want to program a robust Wi-Fi application. This article is supplementary to the Wi-Fi APIs/Examples. It describes the principles of using the Wi-Fi APIs, the limitations of the current Wi-Fi API implementation, and the most common pitfalls in using Wi-Fi. This article also reveals some design details of the Wi-Fi driver. We recommend that you become familiar at least with the following sections: *<ESP32 Wi-Fi API Error Code>*, *<ESP32 Wi-Fi Programming Model>*, and *<ESP32 Wi-Fi Event Description>*.

#### Setting Wi-Fi Compile-time Options

Refer to *<Wi-Fi Menuconfig>*

#### Init Wi-Fi

Refer to *<ESP32 Wi-Fi Station General Scenario>*, *<ESP32 Wi-Fi soft-AP General Scenario>*.

## Start/Connect Wi-Fi

Refer to *<ESP32 Wi-Fi Station General Scenario>*, *<ESP32 Wi-Fi soft-AP General Scenario>*.

## Event-Handling

Generally, it is easy to write code in “sunny-day” scenarios, such as *<SYSTEM\_EVENT\_STA\_START>*, *<SYSTEM\_EVENT\_STA\_CONNECTED>* etc. The hard part is to write routines in “rainy-day” scenarios, such as *<SYSTEM\_EVENT\_STA\_DISCONNECTED>* etc. Good handling of “rainy-day” scenarios is fundamental to robust Wi-Fi applications. Refer to *<ESP32 Wi-Fi Event Description>*, *<ESP32 Wi-Fi Station General Scenario>*, *<ESP32 Wi-Fi soft-AP General Scenario>*

## Write Error-Recovery Routines Correctly at All Times

Just like the handling of “rainy-day” scenarios, a good error-recovery routine is also fundamental to robust Wi-Fi applications. Refer to *<ESP32 Wi-Fi API Error Code>*

### 4.15.4 ESP32 Wi-Fi API Error Code

All of the ESP32 Wi-Fi APIs have well-defined return values, namely, the error code. The error code can be categorized into:

- No errors, e.g. ESP\_ERR\_WIFI\_OK means that the API returns successfully
- Recoverable errors, such as ESP\_ERR\_WIFI\_NO\_MEM, etc.
- Non-recoverable, non-critical errors
- Non-recoverable, critical errors

Whether the error is critical or not depends on the API and the application scenario, and it is defined by the API user.

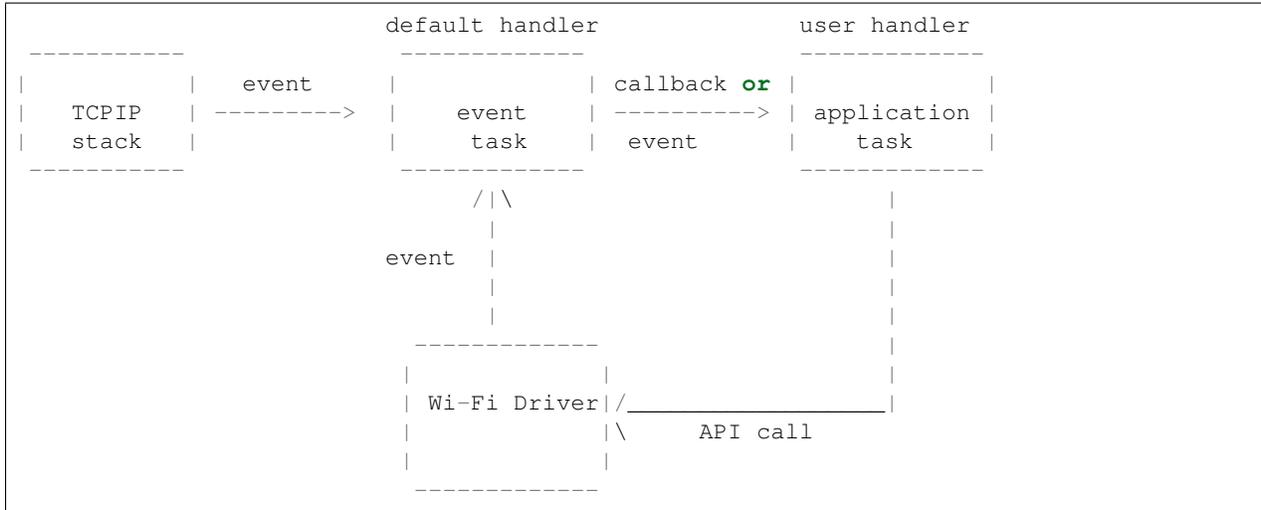
**The primary principle to write a robust application with Wi-Fi API is to always check the error code and write the error-handling code.** Generally, the error-handling code can be used:

- for recoverable errors, in which case you can write a recoverable-error code. For example, when esp\_wifi\_start returns ESP\_ERR\_WIFI\_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, critical errors, in which case “assert” may be a good method for error handling. For example, if esp\_wifi\_set\_mode returns ESP\_ERR\_WIFI\_NOT\_INIT, it means that the Wi-Fi driver is not initialized by esp\_wifi\_init successfully. You can detect this kind of error very quickly in the application development phase.

In esp\_err.h, ESP\_ERROR\_CHECK checks the return values. It is a rather commonplace error-handling code and can be used as the default error-handling code in the application development phase. However, we strongly recommend that the API user writes their own error-handling code.

### 4.15.5 ESP32 Wi-Fi Programming Model

The ESP32 Wi-Fi programming model is depicted as follows:



The Wi-Fi driver can be considered a black box that knows nothing about high-layer code, such as the TCPIP stack, application task, event task, etc. All the Wi-Fi driver can do is receive API calls from the high layer, or post an event-queue to a specified queue which is initialized by API `esp_wifi_init()`.

The event task is a daemon task which receives events from the Wi-Fi driver or from other subsystems, such as the TCPIP stack. The event task will call the default callback function upon receiving the event. For example, upon receiving `SYSTEM_EVENT_STA_CONNECTED`, it will call `tcpip_adapter_start()` to start the DHCP client in its default handler.

An application can register its own event callback function by using API `esp_event_init`. Then, the application callback function will be called after the default callback. Also, if the application does not want to execute the callback in the event task, it needs to post the relevant event to the application task in the application callback function.

The application task (code) generally mixes all these things together: it calls APIs to initialize the system/Wi-Fi and handle the events when necessary.

### 4.15.6 ESP32 Wi-Fi Event Description

#### SYSTEM\_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.

#### SYSTEM\_EVENT\_SCAN\_DONE

The scan-done event is triggered by `esp_wifi_scan_start()` and will arise in the following scenarios:

- The scan is completed, e.g., the target AP is found successfully, or all channels have been scanned.
- The scan is stopped by `esp_wifi_scan_stop()`.
- The `esp_wifi_scan_start()` is called before the scan is completed. A new scan will override the current scan and a scan-done event will be generated.

The scan-done event will not arise in the following scenarios:

- It is a blocked scan.
- The scan is caused by `esp_wifi_connect()`.

Upon receiving this event, the event task does nothing. The application event callback needs to call `esp_wifi_scan_get_ap_num()` and `esp_wifi_scan_get_ap_records()` to fetch the scanned AP list and trigger the Wi-Fi driver to free the internal memory which is allocated during the scan (**do not forget to do this**)! Refer to 'ESP32 Wi-Fi Scan' for a more detailed description.

### SYSTEM\_EVENT\_STA\_START

If `esp_wifi_start()` returns `ESP_OK` and the current Wi-Fi mode is Station or SoftAP+Station, then this event will arise. Upon receiving this event, the event task will initialize the LwIP network interface (`netif`). Generally, the application event callback needs to call `esp_wifi_connect()` to connect to the configured AP.

### SYSTEM\_EVENT\_STA\_STOP

If `esp_wifi_stop()` returns `ESP_OK` and the current Wi-Fi mode is Station or SoftAP+Station, then this event will arise. Upon receiving this event, the event task will release the station's IP address, stop the DHCP client, remove TCP/UDP-related connections and clear the LwIP station `netif`, etc. The application event callback generally does not need to do anything.

### SYSTEM\_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.

### SYSTEM\_EVENT\_STA\_DISCONNECTED

This event can be generated in the following scenarios:

- When `esp_wifi_disconnect()`, or `esp_wifi_stop()`, or `esp_wifi_deinit()`, or `esp_wifi_restart()` is called and the station is already connected to the AP.
- When `esp_wifi_connect()` is called, but the Wi-Fi driver fails to set up a connection with the AP due to certain reasons, e.g. the scan fails to find the target AP, authentication times out, etc.
- When the Wi-Fi connection is disrupted because of specific reasons, e.g., the station continuously loses N beacons, the AP kicks off the station, the AP's authentication mode is changed, etc.

Upon receiving this event, the event task will shut down the station's LwIP `netif` and notify the LwIP task to clear the UDP/TCP connections which cause the wrong status to all sockets. **For socket-based applications, the application callback needs to close all sockets and re-create them, if necessary, upon receiving this event.**

Now, let us consider the following scenario:

- 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 `<SYSTEM_EVENT_STA_DISCONNECTED>` is raised. According to the current implementation, **all TCP connections will be removed and the keep-alive socket will be in a wrong status**. However, since the application designer believes that the network layer should NOT care about this error at the Wi-Fi layer, the application does not close the socket.
- Five seconds later, the Wi-Fi connection is restored because `esp_wifi_connect()` is called in the application event callback function.

- 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.

Generally, if the application has a correct error-handling code, upon receiving `<SYSTEM_EVENT_STA_DISCONNECTED>` the socket can quickly detect the failure without having to wait for 55 seconds. For applications similar to the keep-alive example, we suggest that you close all sockets, once the `<SYSTEM_EVENT_STA_DISCONNECTED>` is received, and that you restart the application when `SYSTEM_EVENT_STA_CONNECTED` arises.

Ideally, the application sockets and the network layer should not be affected, since the Wi-Fi connection only fails temporarily and recovers very quickly. In future IDF releases, we are going to provide a more robust solution for handling events that disrupt Wi-Fi connection, as ESP32's Wi-Fi functionality continuously improves.

### **SYSTEM\_EVENT\_STA\_AUTHMODE\_CHANGE**

This event arises when the AP to which the station is connected changes its authentication mode, e.g., from no auth to WPA. Upon receiving this event, the event task will do nothing. Generally, the application event callback does not need to handle this either.

### **SYSTEM\_EVENT\_STA\_GOT\_IP**

#### **SYSTEM\_EVENT\_AP\_STA\_GOT\_IP6**

This event arises when the DHCP client successfully gets the IP address from the DHCP server. The event means that everything is ready and the application can begin its tasks (e.g., creating sockets).

The IP may be changed because of the following reasons:

- The DHCP client fails to renew/rebind the IP address, and the station's IP is reset to 0.
- The DHCP client rebinds to a different address.
- The static-configured IP address is changed.

The socket is based on the IP address, which means that, if the IP changes, all sockets relating to this IP will become abnormal. Upon receiving this event, the application needs to close all sockets and recreate the application when the IP changes to a valid one.

### **SYSTEM\_EVENT\_AP\_START**

Similar to `<SYSTEM_EVENT_STA_START>`.

### **SYSTEM\_EVENT\_AP\_STOP**

Similar to `<SYSTEM_EVENT_STA_STOP>`.

### **SYSTEM\_EVENT\_AP\_STACONNECTED**

Every time a station is connected to ESP32 SoftAP, the `<SYSTEM_EVENT_AP_STACONNECTED>` will arise. Upon receiving this event, the event task will do nothing, and the application callback can also ignore it. However, you may want to do something, for example, to get the info of the connected STA, etc.

## SYSTEM\_EVENT\_AP\_STADISCONNECTED

This event can happen in the following scenarios:

- The application calls `esp_wifi_disconnect()`, or `esp_wifi_deinit_sta()`, to manually disconnect the station.
- The Wi-Fi driver kicks off the station, e.g. because the SoftAP has not received any packets in the past five minutes, etc.
- The station kicks off the SoftAP.

When this event happens, the event task will do nothing, but the application event callback needs to do something, e.g., close the socket which is related to this station, etc.

## SYSTEM\_EVENT\_AP\_PROBEREQRECVED

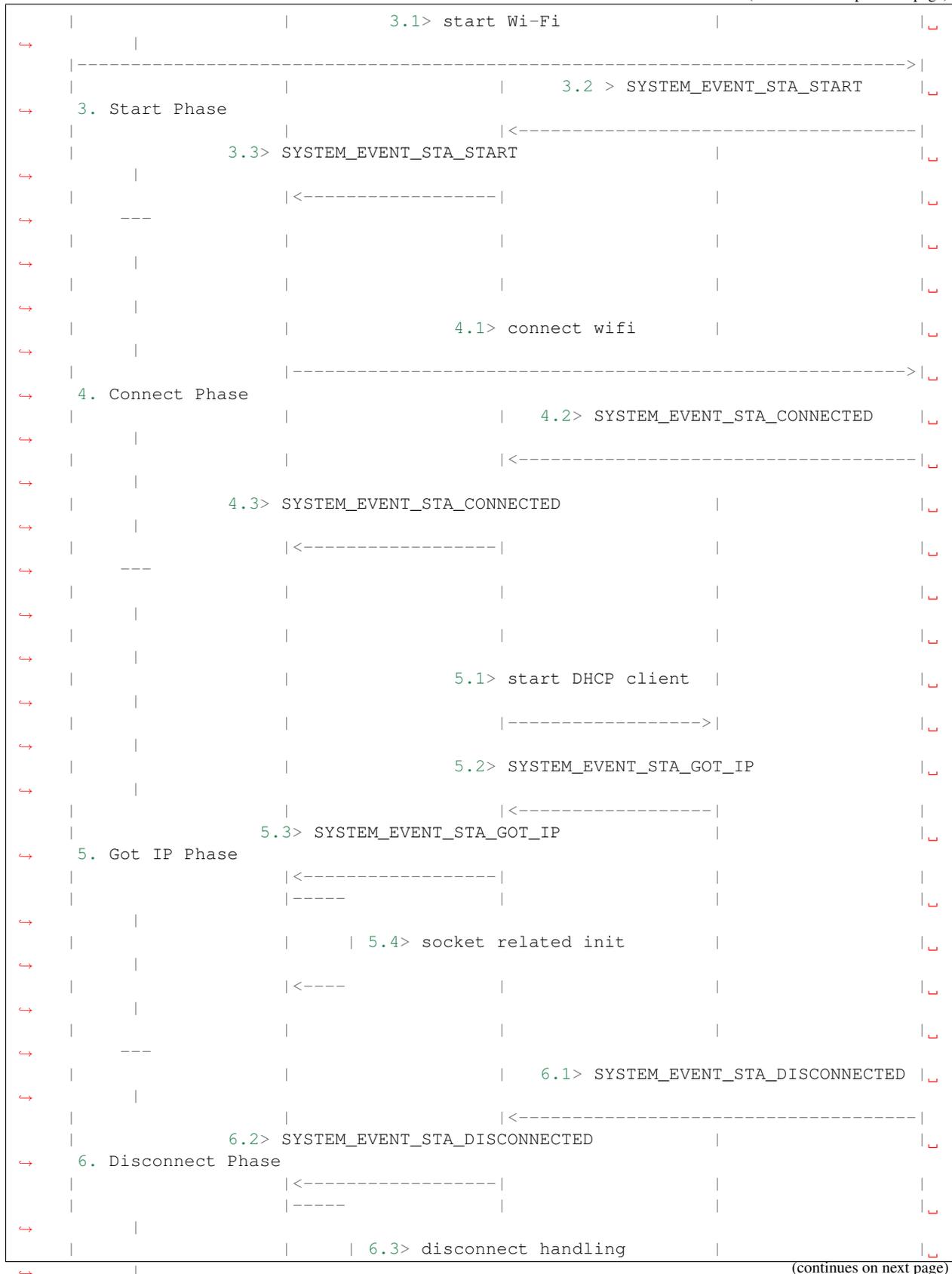
Currently, the ESP32 implementation will never generate this event. It may be removed in future releases.

### 4.15.7 ESP32 Wi-Fi Station General Scenario

Below is a “big scenario” which describes some small scenarios in Station mode:

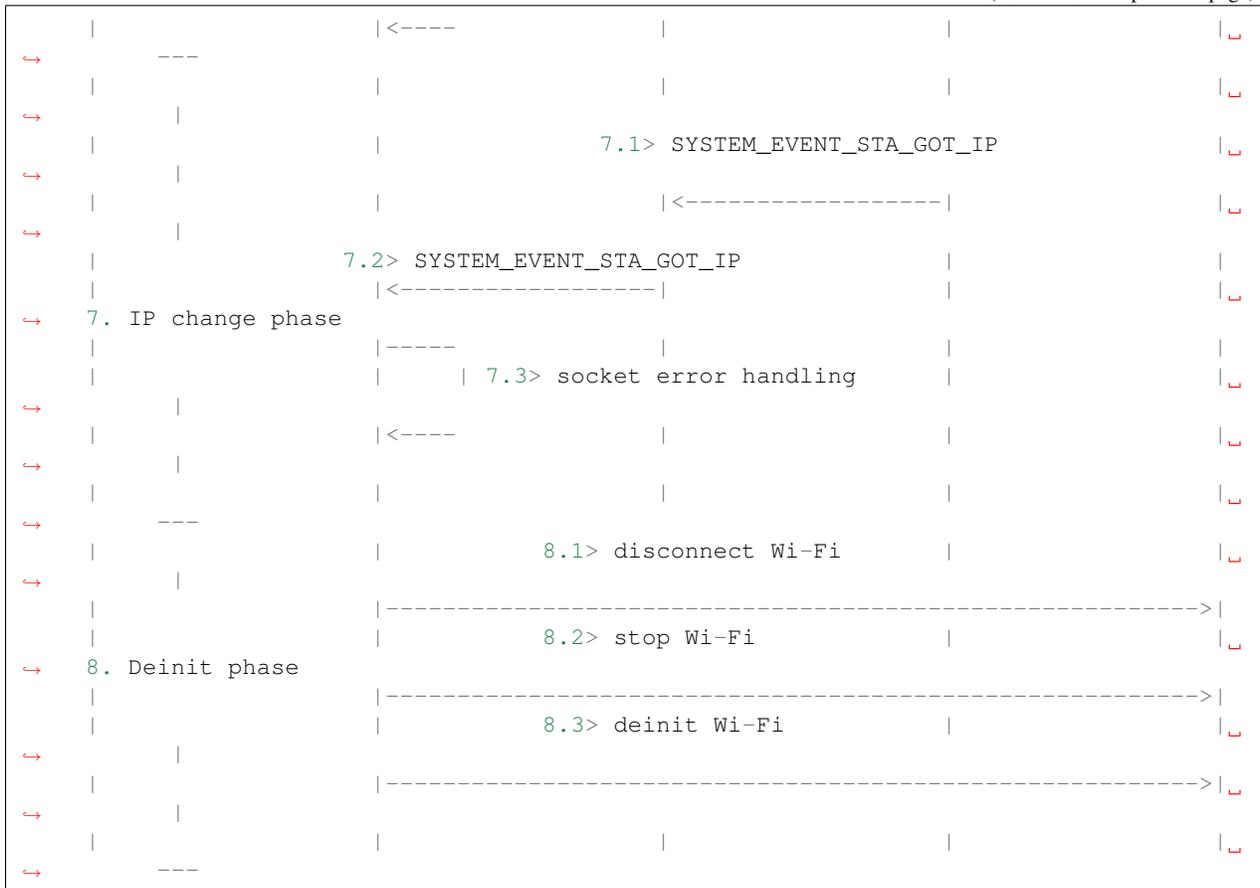


(continued from previous page)



(continues on next page)

(continued from previous page)



## 1. Wi-Fi/LwIP Init Phase

- s1.1: The main task calls `tcpip_adapter_init()` to create an LwIP core task and initialize LwIP-related work.
- s1.2: The main task calls `esp_event_loop_init()` to create a system Event task and initialize an application event's callback function. In the scenario above, the application event's callback function does nothing but relaying the event to the application task.
- s1.3: The main task calls `esp_wifi_init()` to create the Wi-Fi driver task and initialize the Wi-Fi driver.
- s1.4: The main task calls OS API to create the application task.

Step 1.1~1.4 is a recommended sequence that initializes a Wi-Fi/LwIP-based application. However, it is **NOT** a must-follow sequence, which means that you can create the application task in step 1.1 and put all other initializations in the application task. Moreover, you may not want to create the application task in the initialization phase if the application task depends on the sockets. Rather, you can defer the task creation until the IP is obtained.

## 2. Wi-Fi Configuration Phase

Once the Wi-Fi driver is initialized, you can start configuring the Wi-Fi driver. In this scenario, the mode is Station, so you may need to call `esp_wifi_set_mode(WIFI_MODE_STA)` to configure the Wi-Fi mode as Station. You can call other `esp_wifi_set_xxx` APIs to configure more settings, such as the protocol mode, country code, bandwidth, etc. Refer to [<ESP32 Wi-Fi Configuration>](#).

Generally, we configure the Wi-Fi driver before setting up the Wi-Fi connection, but this is **NOT** mandatory, which means that you can configure the Wi-Fi connection anytime, provided that the Wi-Fi driver is initialized successfully. However, if the configuration does not need to change after the Wi-Fi connection is set up, you should configure the Wi-Fi driver at this stage, because the configuration APIs (such as `esp_wifi_set_protocol`) will cause the Wi-Fi to reconnect, which may not be desirable.

If the Wi-Fi NVS flash is enabled by menuconfig, all Wi-Fi configuration in this phase, or later phases, will be stored into flash. When the board powers on/reboots, you do not need to configure the Wi-Fi driver from scratch. You only need to call `esp_wifi_get_xxx` APIs to fetch the configuration stored in flash previously. You can also configure the Wi-Fi driver if the previous configuration is not what you want.

### 3. Wi-Fi Start Phase

- s3.1: Call `esp_wifi_start` to start the Wi-Fi driver.
- s3.2: The Wi-Fi driver posts `<SYSTEM_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 `<SYSTEM_EVENT_STA_START>` to the application task. We recommend that you call `esp_wifi_connect()`. However, you can also call `esp_wifi_connect()` in other phrases after the `<SYSTEM_EVENT_STA_START>` arises.

### 4. Wi-Fi Connect Phase

- s4.1: Once `esp_wifi_connect()` is called, the Wi-Fi driver will start the internal scan/connection process.
- s4.2: If the internal scan/connection process is successful, the `<SYSTEM_EVENT_STA_CONNECTED>` will be generated. In the event task, it starts the DHCP client, which will finally trigger the DHCP process.
- s4.3: In the above-mentioned scenario, the application event callback will relay the event to the application task. Generally, the application needs to do nothing, and you can do whatever you want, e.g., print a log, etc.

In step 4.2, the Wi-Fi connection may fail because, for example, the password is wrong, the AP is not found, etc. In a case like this, `<SYSTEM_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 `<SYSTEM_EVENT_STA_GOT_IP>` will arise and the event task will perform common handling.
- s5.3: In the application event callback, `<SYSTEM_EVENT_STA_GOT_IP>` is relayed to the application task. For LwIP-based applications, this event is very special and means that everything is ready for the application to begin its tasks, e.g. creating the TCP/UDP socket, etc. A very common mistake is to initialize the socket before `<SYSTEM_EVENT_STA_GOT_IP>` is received. **DO NOT start the socket-related work before the IP is received.**

### 6. Wi-Fi Disconnect Phase

- s6.1: When the Wi-Fi connection is disrupted, e.g. because the AP is powered off, the RSSI is poor, etc., `<SYSTEM_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 `<SYSTEM_EVENT_STA_DISCONNECTED>` to the application task. We recommend that `esp_wifi_connect()` be called to reconnect the Wi-Fi, close all sockets and re-create them if necessary. Refer to `<SYSTEM_EVENT_STA_DISCONNECTED>`.

## 7. Wi-Fi IP Change Phase

- s7.1: If the IP address is changed, the `<SYSTEM_EVENT_STA_GOT_IP>` will arise.
- 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.

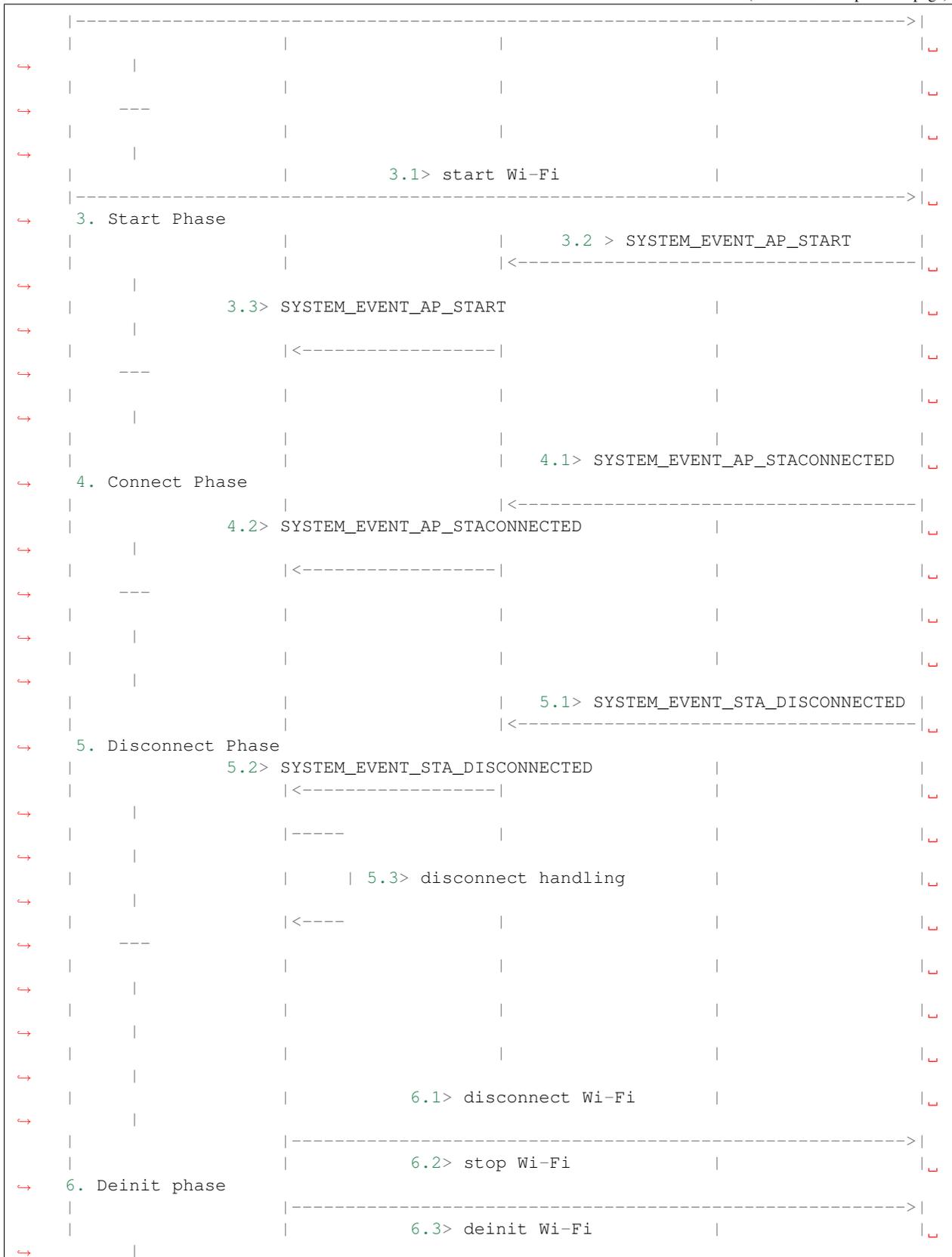
### 4.15.8 ESP32 Wi-Fi soft-AP General Scenario

Below is a “big scenario” which describes some small scenarios in Soft-AP mode:



(continues on next page)

(continued from previous page)



(continues on next page)

(continued from previous page)



### 4.15.9 ESP32 Wi-Fi Scan

Currently, the `esp_wifi_scan_start()` API is supported only in Station or Station+SoftAP mode.

#### Scan Type

Mode	Description
Active Scan	Scan by sending a probe request. The default scan is an active scan.
Passive Scan	No probe request is sent out. Just switch to the specific channel and wait for a beacon. Application can enable it via the <code>scan_type</code> field of <code>wifi_scan_config_t</code> .
Foreground Scan	This scan is applicable when there is no Wi-Fi connection in Station mode. Foreground or background scanning is controlled by the Wi-Fi driver and cannot be configured by the application.
Background Scan	This scan is applicable when there is a Wi-Fi connection in Station mode or in Station+SoftAP mode. Whether it is a foreground scan or background scan depends on the Wi-Fi driver and cannot be configured by the application.
All-Channel Scan	It scans all of the channels. If the <code>channel</code> field of <code>wifi_scan_config_t</code> is set to 0, it is an all-channel scan.
<b>Specific Channel Scan</b>	It scans specific channels only. If the <code>channel</code> field of <code>wifi_scan_config_t</code> set to 1, it is a specific-channel scan.

The scan modes in above table can be combined arbitrarily, so we totally have 8 different scans:

- All-Channel Background Active Scan
- All-Channel Background Passive Scan
- All-Channel Foreground Active Scan
- All-Channel Foreground Passive Scan
- Specific-Channel Background Active Scan
- Specific-Channel Background Passive Scan
- Specific-Channel Foreground Active Scan
- Specific-Channel Foreground Passive Scan









## Scan 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.

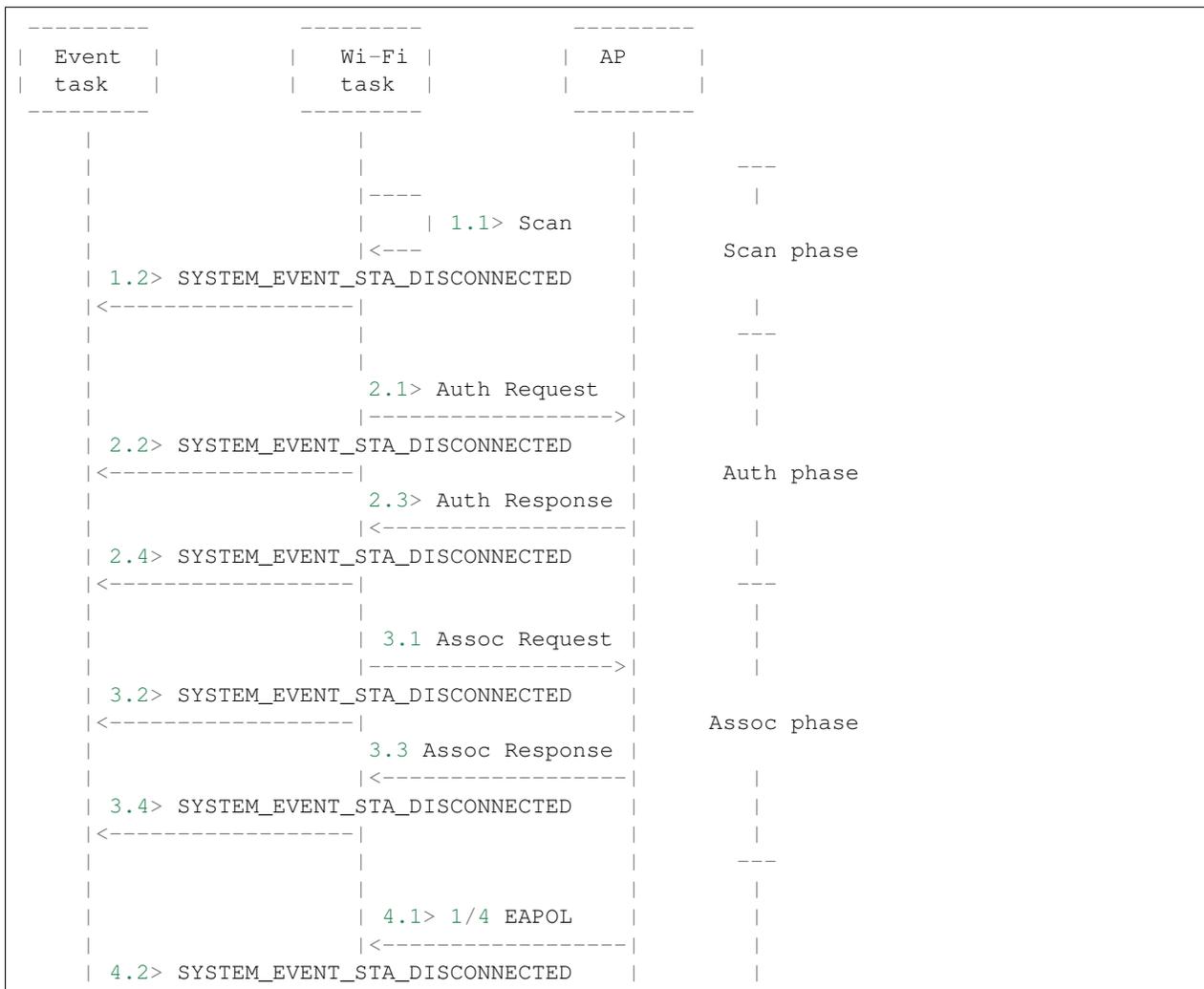
## Parallel Scan

Two application tasks may call `esp_wifi_scan()` 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, in IDF2.1, the Wi-Fi driver does not support parallel scans adequately. As a result, a parallel scan should be avoided.** The parallel scan will be enhanced in future releases, as the ESP32's Wi-Fi functionality improves continuously.

### 4.15.10 ESP32 Wi-Fi Station Connecting Scenario

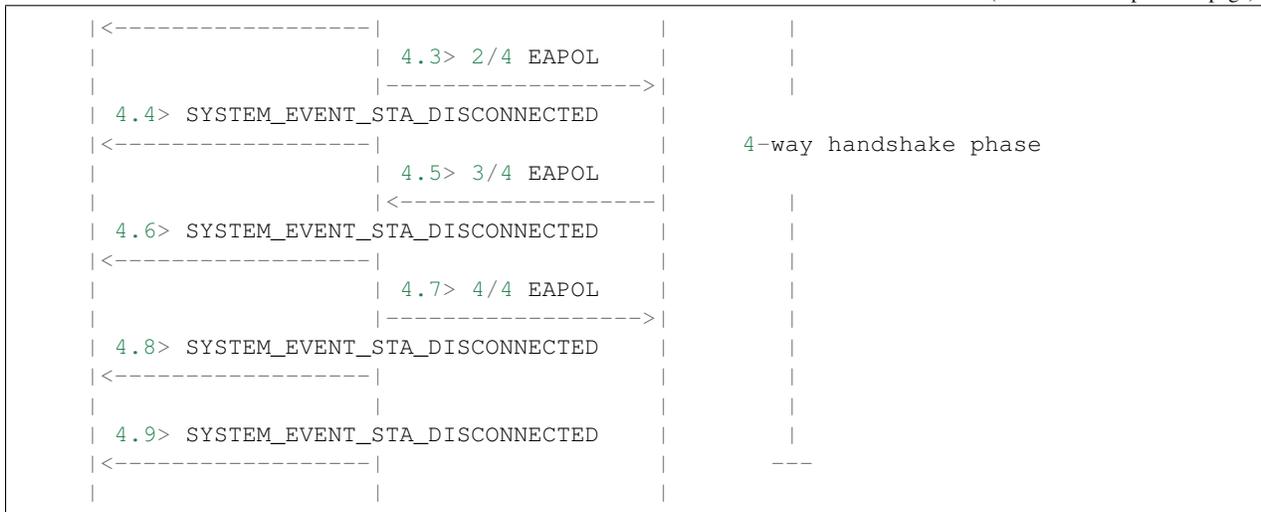
Generally, the application does not need to care about the connecting process. Below is a brief introduction to the process for those who are really interested.

Scenario:



(continues on next page)

(continued from previous page)



### 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, *<SYSTEM\_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, *<SYSTEM\_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 *<SYSTEM\_EVENT\_STA\_DISCONNECTED>* arises, while the reason-code is WIFI\_REASON\_AUTH\_FAIL or the reasons specified by the soft-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, *<SYSTEM\_EVENT\_STA\_DISCONNECTED>* will arise and the reason-code will be WIFI\_REASON\_ASSOC\_EXPIRE. Refer to *<Wi-Fi Reason Code>*.
- s3.3, The association response is received and the association timer is stopped.
- s3.4, The AP rejects the association in the response and *<SYSTEM\_EVENT\_STA\_DISCONNECTED>* arises, while the reason-code is the one specified in the association response. Refer to *<Wi-Fi Reason Code>*.

## Four-way Handshake Phase

- s4.1, The four-way handshake is sent out and the association timer is enabled.
- s4.2, If the association response is not received before the association timer times out, `<SYSTEM_EVENT_STA_DISCONNECTED>` will arise and the reason-code will be `WIFI_REASON_ASSOC_EXPIRE`. Refer to `<Wi-Fi Reason Code>`.
- s4.3, The association response is received and the association timer is stopped.
- s4.4, The AP rejects the association in the response and `<SYSTEM_EVENT_STA_DISCONNECTED>` arises and the reason-code will be the one specified in the association response. Refer to `<Wi-Fi Reason Code>`.

## Wi-Fi Reason Code

The table below shows the reason-code defined in ESP32. The first column is the macro name defined in `esp_wifi_types.h`. The common prefix `WIFI_REASON` is removed, which means that `UNSPECIFIED` actually stands for `WIFI_REASON_UNSPECIFIED` and so on. The second column is the value of the reason. The third column is the standard value to which this reason is mapped in section 8.4.1.7 of IEEE 802.11-2012. (For more information, refer to the standard mentioned above.) The last column is a description of the reason.

Reason code	ESP32 value	Mapped To Standard Value	Description
UNSPECIFIED	1	1	Generally, it means an internal failure, e.g., the memory runs out, the internal TX fails, or the reason is received from the remote side, etc.
AUTH_EXPIRE	2	2	<p>The previous authentication is no longer valid.</p> <p>For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> <li>• auth is timed out</li> <li>• the reason is received from the soft-AP.</li> </ul> <p>For the ESP32 SoftAP, this reason is reported when:</p> <ul style="list-style-type: none"> <li>• the soft-AP has not received any packets from the station in the past five minutes.</li> <li>• the soft-AP is stopped by calling esp_wifi_stop().</li> <li>• the station is deauthenticated by calling esp_wifi_deauth_sta()</li> </ul>
AUTH_LEAVE	3	3	<p>De-authenticated, because the sending STA is leaving (or has left).</p> <p>For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> <li>• it is received from the soft-AP.</li> </ul>
ASSOC_EXPIRE	4	4	<p>Disassociated due to inactivity.</p> <p>For the ESP32 Station, this reason is reported when:</p> <ul style="list-style-type: none"> <li>• it is received from the soft-AP.</li> </ul> <p>For the ESP32 Soft-AP, this reason is reported when:</p> <ul style="list-style-type: none"> <li>• the soft-AP has not received any packets from the station in the past five min-</li> </ul>
848			<p><b>Chapter 4. API Guides</b></p> <ul style="list-style-type: none"> <li>• the soft-AP is stopped by calling esp_wifi_stop().</li> </ul>

## 4.15.11 ESP32 Wi-Fi Configuration

All configurations will be stored into flash when the Wi-Fi NVS is enabled; otherwise, refer to <*Wi-Fi NVS Flash*>.

### Wi-Fi Mode

Call `esp_wifi_set_mode()` to set the Wi-Fi mode.

Mode	Description
WIFI_MODE_NULL	<b>Null mode:</b> in this mode, the internal data struct is not allocated to the station and the soft-AP, while both the station and soft-AP interfaces are not initialized for RX/TX Wi-Fi data. Generally, this mode is used for Sniffer, or when you only want to stop both the STA and the AP without calling <code>esp_wifi_deinit()</code> to unload the whole Wi-Fi driver.
WIFI_MODE_STA	<b>Station mode:</b> in this mode, <code>esp_wifi_start()</code> will init the internal station data, while the station's interface is ready for the RX and TX Wi-Fi data. After <code>esp_wifi_connect()</code> is called, the STA will connect to the target AP.
WIFI_MODE_AP	<b>Soft-AP mode:</b> in this mode, <code>esp_wifi_start()</code> will init the internal soft-AP data, while the soft-AP's interface is ready for RX/TX Wi-Fi data. Then, the Wi-Fi driver starts broadcasting beacons, and the soft-AP is ready to get connected to other stations.
WIFI_MODE_APSTA	<b>Station and Soft-AP coexistence mode:</b> in this mode, <code>esp_wifi_start()</code> will simultaneously init both the station and the soft-AP. This is done in station mode and soft-AP mode. Please note that the channel of the external AP, which the ESP32 Station is connected to, has higher priority over the ESP32 Soft-AP channel. Refer to <i>Wi-Fi Channel Management</i> .

### Station Basic Configuration

API `esp_wifi_set_config()` can be used to configure the station. The table below describes the fields in detail.

Field	Description
ssid	This is the SSID of the target AP, to which the station wants to connect to.
password	Password of the target AP
bssid	If <code>bssid_set</code> 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 as the "bssid" field.
bssid	This is valid only when <code>bssid_set</code> is 1; see field "bssid_set".
channel	If the channel is 0, the station scans the channel 1~N to search for the target AP; otherwise, the station starts by scanning the channel whose value is the same as that of the "channel" field, and then scans others to find the target AP. If you do not know which channel the target AP is running on, set it to 0.

### Soft-AP Basic Configuration

API `esp_wifi_set_config()` can be used to configure the soft-AP. The table below describes the fields in detail.

Field	Description
ssid	SSID of soft-AP; if the ssid[0] is 0xFF and ssid[1] is 0xFF, the soft-AP defaults the SSID to ESP_aabcc, where “aabcc” is the last three bytes of the soft-AP MAC.
password	Password of soft-AP; if the auth mode is WIFI_AUTH_OPEN, this field will be ignored.
ssid_len	Length of SSID; if ssid_len is 0, check the SSID until there is a termination character. If ssid_len > 32, change it to 32; otherwise, set the SSID length according to ssid_len.
channel	Channel of soft-AP; if the channel is out of range, the Wi-Fi driver defaults the channel to channel 1. So, please make sure the channel is within the required range. For more details, refer to <Channel Range>.
auth-mode	Auth mode of ESP32 soft-AP; currently, ESP32 Wi-Fi does not support AUTH_WEP. If the authmode is an invalid value, soft-AP defaults the value to WIFI_AUTH_OPEN.
ssid_hidden	If ssid_hidden is 1, soft-AP does not broadcast the SSID; otherwise, it does broadcast the SSID.
max_connection	Currently, ESP32 Wi-Fi supports up to 10 Wi-Fi connections. If max_connection > 10, soft-AP defaults the value to 10.
beacon_interval	Beacon interval; the value is 100 ~ 60000 ms, with default value being 100 ms. If the value is out of range, soft-AP defaults it to 100 ms.

## Wi-Fi Protocol Mode

Currently, the IDF supports the following protocol modes:

Protocol Mode	Description
802.11B	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B) to set the station/soft-AP to 802.11B-only mode.
802.11BG	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G) to set the station/soft-AP to 802.11BG mode.
802.11BGN	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G WIFI_PROTOCOL_11N) to set the station/ soft-AP to BGN mode.
802.11BGNLR	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_11B WIFI_PROTOCOL_11G WIFI_PROTOCOL_11N WIFI_PROTOCOL_LR) to set the station/soft-AP to BGN and the Espressif-specific mode.
802.11LR	Call esp_wifi_set_protocol(ifx, WIFI_PROTOCOL_LR) to set the station/soft-AP only to the Espressif-specific mode. <b>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 soft-AP are connected to an ESP32 device</b>

## Wi-Fi Channel Management

### Channel Range

Call esp\_wifi\_set\_country() to set the country code which limits the channel range.

Country	Channel Range
China	1,2,3 ... 14
Japan	1,2,3 ... 14
USA	1,2,3 ... 11
Europe	1,2,3 ... 13

## Home Channel

In soft-AP mode, the home channel is defined as that of the soft-AP channel. In Station mode, the home channel is defined as the channel of the AP to which the station is connected. In Station+SoftAP mode, the home channel of soft-AP and station must be the same. If the home channels of Station and Soft-AP are different, the station's home channel is always in priority. Take the following as an example: at the beginning, the soft-AP is on channel 6, then the station connects to an AP whose channel is 9. Since the station's home channel has a higher priority, the soft-AP needs to switch its channel from 6 to 9 to make sure that both station and soft-AP have the same home channel.

## Wi-Fi Vendor IE Configuration

By default, all Wi-Fi management frames are processed by the Wi-Fi driver, and the application does not need to care about them. Some applications, however, may have to handle the beacon, probe request, probe response and other management frames. For example, if you insert some vendor-specific IE into the management frames, it is only the management frames which contain this vendor-specific IE that will be processed. In ESP32, `esp_wifi_set_vendor_ie()` and `esp_wifi_set_vendor_ie_cb()` are responsible for this kind of tasks.

### 4.15.12 ESP32 Wi-Fi Power-saving Mode

Currently, ESP32 Wi-Fi supports the Modem-sleep mode which refers to WMM (Wi-Fi Multi Media) power-saving mode in the IEEE 802.11 protocol. If the Modem-sleep mode is enabled and the Wi-Fi enters a sleep state, then, RF, PHY and BB are turned off in order to reduce power consumption. Modem-sleep mode works in Station-only mode and the station must be connected to the AP first.

Call `esp_wifi_set_ps(WIFI_PS_MODEM)` to enable Modem-sleep mode after calling `esp_wifi_init()`. About 10 seconds after the station connects to the AP, Modem-sleep will start. When the station disconnects from the AP, Modem-sleep will stop.

### 4.15.13 ESP32 Wi-Fi Connect Crypto

Now ESP32 have two group crypto functions can be used when do wifi connect, one is the original functions, the other is optimized by ESP hardware: 1. Original functions which is the source code used in the folder `components/wpa_supplicant/src/crypto` function; 2. The optimized functions is in the folder `components/wpa_supplicant/src/fast_crypto`, these function used the hardware crypto to make it faster than origin one, the type of function's name add *fast\_* to distinguish with the original one. For example, the API `aes_wrap()` is used to encrypt frame information when do 4 way handshake, the `fast_aes_wrap()` has the same result but can be faster.

Two groups of crypto function can be used when register in the `wpa_crypto_funcs_t`, `wpa2_crypto_funcs_t` and `wps_crypto_funcs_t` structure, also we have given the recommend functions to register in the `fast_crypto_ops.c`, you can register the function as the way you need, however what should make action is that the `crypto_hash_xxx` function and `crypto_cipher_xxx` function need to register with the same function to operation. For example, if you register `crypto_hash_init()` function to initialize the `esp_crypto_hash` structure, you need use the `crypto_hash_update()` and `crypto_hash_finish()` function to finish the operation, rather than `fast_crypto_hash_update()` or `fast_crypto_hash_finish()`.

### 4.15.14 ESP32 Wi-Fi Throughput

The table below shows the best throughput results we got in Espressif's lab and in a shield box.

Type/Throughput	Air In Lab	Shield-box
Raw 802.11 Packet RX	N/A	<b>130 MBit/sec</b>
Raw 802.11 Packet TX	N/A	<b>130 MBit/sec</b>
UDP RX	30 MBit/sec	80 MBit/sec
UDP TX	30 MBit/sec	80 MBit/sec
TCP RX	20 MBit/sec	25 MBit/sec
TCP TX	20 MBit/sec	25 MBit/sec

The throughput result heavily depends on hardware and software configurations, such as CPU frequency, memory configuration, or whether the CPU is running in dual-core mode, etc. The table below shows the configurations with which we got the above-mentioned throughput results. In ESP32 IDF, the default configuration is based on “very conservative” calculations, so if you want to get the best throughput result, the first thing you need to do is to adjust the relevant configurations.

Type	Value	How to configure
CPU Core Mode	Dual Core	Menuconfig
CPU Frequency	240 MHz	Menuconfig
Static RX Buffer	15	Menuconfig
Dynamic RX Buffer	Unlimited	Menuconfig
Dynamic TX Buffer	Unlimited	Menuconfig
TCP RX Window	12*1460 Bytes	Release 2.1/2.0 and earlier: TCP_WND_DEFAULT in lwipopts.h After the 2.1 Release: Menuconfig
TCP TX Window	12*1460 Bytes	Release 2.1/2.0 and earlier: TCP_SND_BUF_DEFAULT in lwipopts.h After the 2.1 Release: Menuconfig
TCP RX MBOX	12	Release 2.1/2.0 and earlier: DEFAULT_TCP_RECVMBOX_SIZE in lwipopts.h After the 2.1 Release: Menuconfig
RX BA Window	9~16	Release 2.1/2.0 and earlier: not configurable After the 2.1 Release: Menuconfig
TX BA Window	9~16	Release 2.1/2.0 and earlier: not configurable After the 2.1 Release: Menuconfig

Once you adjust the configurations, you can then run your own test code to test the performance. You can also run the iperf example to test the performance. However, the iperf example is not provided in release 2.1 and earlier ones, but will be so in the upcoming release. Those who really care about the performance should seek support from Espressif directly, so that we can provide them with the iperf version bin for their testing.

If you decide to modify some of the configurations in order to gain better throughput for your application, please consider the memory usage very carefully. For a more detailed description, refer to [<Wi-Fi Buffer Usage>](#) and [<Wi-Fi Buffer Configure>](#).

#### 4.15.15 Wi-Fi 80211 Packet Send

**Important notes:** The API `esp_wifi_80211_tx` is not available in IDF 2.1, but will be so in the upcoming release.

The `esp_wifi_80211_tx` API can be used to:

- Send the beacon, probe request, probe response, action frame.

- Send the QoS and non-QoS data frame.

It cannot be used for sending cryptographic frames.

### Parameters of esp\_wifi\_80211\_tx

Parameter	Description
ifx	Wi-Fi interface ID: 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 802.11 buffer. For building the correct buffer, refer to the following sections: If the buffer is wrong, or violates the Wi-Fi driver's restrictions, the API returns ESP_ERR_WIFI_ARG or results in unexpected behavior. <b>Please read the following section carefully to make sure you understand the restrictions on encapsulating the buffer.</b>
len	The length must be $\leq 1500$ ; otherwise, the API will return ESP_ERR_WIFI_ARG.
en_sys_seq	If en_sys_seq is true, it means that the Wi-Fi driver will rewrite the sequence number in the buffer; otherwise, it will not rewrite the 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 should be true. For more details, read the following section about the sequence configuration.

### Preconditions of Using esp\_wifi\_80211\_tx

- The Wi-Fi mode is Station, or SoftAP, or Station+SoftAP.
- Either esp\_wifi\_set\_promiscuous(true), or esp\_wifi\_start(), or both of these APIs return ESP\_ERR\_WIFI\_OK. This is because we need to make sure that Wi-Fi hardware is initialized before esp\_wifi\_80211\_tx() is called. In ESP32, both esp\_wifi\_set\_promiscuous(true) and esp\_wifi\_start() can trigger the initialization of Wi-Fi hardware.
- The parameters of esp\_wifi\_80211\_tx are hereby correctly provided.

### Side-Effects to Avoid in Different Scenarios

Theoretically, if we do not consider the side-effects the API imposes on the Wi-Fi driver or other stations/soft-APs, we can send a raw 802.11 packet over the air, with any destination MAC, any source MAC, any BSSID, or any other type of packet. However, robust/useful applications should avoid such side-effects. The table below provides some tips/recommendations on how to avoid the side-effects of esp\_wifi\_80211\_tx in different scenarios.

Scenario	Description
<p>NO_CONN_MGMT</p> <ul style="list-style-type: none"> <li>• No Wi-Fi connection</li> <li>• Can send management frame</li> </ul>	<p>In this scenario, no Wi-Fi connection is set up, so there are no side-effects on the Wi-Fi driver. If <code>en_sys_seq==true</code>, the Wi-Fi driver is responsible for the sequence control. If <code>en_sys_seq==false</code>, the application needs to ensure that the buffer has the correct sequence.</p> <p>Theoretically, the MAC address can be any address. However, this may impact other stations/soft-APs with the same MAC/BSSID.</p> <p>Side-effect example#1 The application calls <code>esp_wifi_80211_tx</code> to send a beacon with BSSID == <code>mac_x</code> in SoftAP mode, but the <code>mac_x</code> is not the MAC of the SoftAP interface. Moreover, there is another soft-AP, say “other-AP”, whose bssid is <code>mac_x</code>. If this happens, an “unexpected behavior” may occur, because the stations which connect to the “other-AP” cannot figure out whether the beacon is from the “other-AP” or the <code>esp_wifi_80211_tx</code>.</p> <p>To avoid the above-mentioned side-effects, we recommend that:</p> <ul style="list-style-type: none"> <li>• If <code>esp_wifi_80211_tx</code> is called in Station mode, the first MAC should be a multicast MAC or the exact target-device’s MAC, while the second MAC should be that of the station interface.</li> <li>• If <code>esp_wifi_80211_tx</code> is called in SoftAP 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 soft-AP interface.</li> </ul> <p>The recommendations above are only for avoiding side-effects and can be ignored when there are good reasons for doing this.</p>
<p>NO_CONN_NON_QOS</p> <ul style="list-style-type: none"> <li>• No Wi-Fi connection</li> <li>• Can send non-QoS frame</li> </ul>	<p>Same as in NO_CONN_MGMT</p>
<p>NO_CONN_QOS</p> <ul style="list-style-type: none"> <li>• No Wi-Fi connection</li> <li>• Can send QoS frame</li> </ul>	<p>Same as in NO_CONN_MGMT</p>
<p>CONN_MGMT</p> <ul style="list-style-type: none"> <li>• Have Wi-Fi connection</li> <li>• Send management frame only</li> </ul>	<p>When the Wi-Fi connection is already set up, and the sequence is controlled by the application, the latter may impact the sequence control of the Wi-Fi connection, as a whole. So, the recommendation is that <code>en_sys_seq</code> be true. The MAC-address recommendations in the NO_CONN_MGMT scenario also apply to the CONN_MGMT scenario.</p>
<p>CONN_NON_QOS</p> <ul style="list-style-type: none"> <li>• Have Wi-Fi connection</li> <li>• Can send non-QoS frame</li> </ul>	<p>Generally, we should use a socket API, instead of this one, in order to send the data frame when the Wi-Fi connection is already set up.</p> <p>However, if you have any special reasons for using this particular API, then <code>en_sys_seq</code> must be true; otherwise, you may impact the internal sequence control of the Wi-Fi connection described in CONN_MGMT. The MAC-address recommendations in the NO_CONN_MGMT</p>
<p>854</p>	<p>scenario also apply to the CONN_MGMT scenario. <b>Chapter 4. API Guides</b></p>
<p>CONN_QOS</p> <ul style="list-style-type: none"> <li>• Have Wi-Fi connection</li> <li>• Can send non-QoS frame</li> </ul>	<p>Same as in CONN_NON_QOS</p>

### 4.15.16 Wi-Fi Sniffer Mode

The Wi-Fi sniffer mode can be enabled by `esp_wifi_set_promiscuous()`. If the sniffer mode is enabled, the following packets **can** be dumped to the application:

- 802.11 Management frame
- 802.11 Data frame, including MPDU, AMPDU, AMSDU, etc.
- 802.11 MIMO frame, for MIMO frame, the sniffer only dumps the length of the frame.

The following packets will **NOT** be dumped to the application:

- 802.11 Control frame
- 802.11 error frame, such as the frame with a CRC error, etc.

For frames that the sniffer **can** dump, the application can additionally decide which specific type of packets can be filtered to the application by using `esp_wifi_set_promiscuous_filter()`. By default, it will filter all 802.11 data and management frames to the application.

The Wi-Fi sniffer mode can be enabled in the Wi-Fi mode of `WIFI_MODE_NULL`, or `WIFI_MODE_STA`, or `WIFI_MODE_AP`, or `WIFI_MODE_APSTA`. In other words, the sniffer mode is active when the station is connected to the soft-AP, or when the soft-AP has a Wi-Fi connection. Please note that the sniffer has a **great impact** on the throughput of the station or soft-AP Wi-Fi connection. Generally, we should **NOT** enable the sniffer, when the station/soft-AP Wi-Fi connection experiences heavy traffic unless we have special reasons.

Another noteworthy issue about the sniffer is the callback `wifi_promiscuous_cb_t`. The callback will be called directly in the Wi-Fi driver task, so if the application has a lot of work to do for each filtered packet, the recommendation is to post an event to the application task in the callback and defer the real work to the application task.

### 4.15.17 Wi-Fi Buffer Usage

This section is only about the dynamic buffer configuration.

#### Why Buffer Configuration Is Important

**In order to get a robust, high-performance system, we need to consider the memory usage/configuration very carefully, because**

- the available memory in ESP32 is limited.
- currently, the default type of buffer in LwIP and Wi-Fi drivers is “dynamic”, **which means that both the LwIP and Wi-Fi share memory with the application**. Programmers should always keep this in mind; otherwise, they will face a memory issue, such as “running out of heap memory”.
- it is very dangerous to run out of heap memory, as this will cause ESP32 an “undefined behavior”. Thus, enough heap memory should be reserved for the application, so that it never runs out of it.
- the Wi-Fi throughput heavily depends on memory-related configurations, such as the TCP window size, Wi-Fi RX/TX dynamic buffer number, etc. Refer to *<ESP32 Wi-Fi Throughput>*.
- the peak heap memory that the ESP32 LwIP/Wi-Fi may consume depends on a number of factors, such as the maximum TCP/UDP connections that the application may have, etc.
- the total memory that the application requires is also an important factor when considering memory configuration.

Due to these reasons, there is not a good-for-all application configuration. Rather, we have to consider memory configurations separately for every different application.

## Dynamic vs. Static Buffer

The default type of buffer in LwIP and 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 LwIP/Wi-Fi.

## Peak LwIP Dynamic Buffer

The default type of LwIP buffer is “dynamic”, and this section considers the dynamic buffer only. The peak heap memory that LwIP consumes is the **theoretically-maximum memory** that the LwIP driver consumes. Generally, the peak heap memory that the LwIP consumes depends on:

- the memory required to create a UDP connection: `lwip_udp_conn`
- the memory required to create a TCP connection: `lwip_tcp_conn`
- the number of UDP connections that the application has: `lwip_udp_con_num`
- the number of TCP connections that the application has: `lwip_tcp_con_num`
- the TCP TX window size: `lwip_tcp_tx_win_size`
- the TCP RX window size: `lwip_tcp_rx_win_size`

**So, the peak heap memory that the LwIP consumes can be calculated with the following formula:**

$$\text{lwip\_dynamic\_peek\_memory} = (\text{lwip\_udp\_con\_num} * \text{lwip\_udp\_conn}) + (\text{lwip\_tcp\_con\_num} * (\text{lwip\_tcp\_tx\_win\_size} + \text{lwip\_tcp\_rx\_win\_size} + \text{lwip\_tcp\_conn}))$$

Some TCP-based applications need only one TCP connection. However, they may choose to close this TCP connection and create a new one when an error (such as a sending failure) occurs. This may result in multiple TCP connections existing in the system simultaneously, because it may take a long time for a TCP connection to close, according to the TCP state machine (refer to RFC793).

## Peak Wi-Fi Dynamic Buffer

The Wi-Fi driver supports several types of buffer (refer to *Wi-Fi Buffer Configure*). However, this section is about the usage of the dynamic Wi-Fi buffer only. The peak heap memory that Wi-Fi consumes is the **theoretically-maximum memory** that the Wi-Fi driver consumes. Generally, the peak memory depends on:

- the number of dynamic rx buffers that are configured: `wifi_rx_dynamic_buf_num`
- the number of dynamic tx buffers that are configured: `wifi_tx_dynamic_buf_num`
- the maximum packet size that the Wi-Fi driver can receive: `wifi_rx_pkt_size_max`
- the maximum packet size that the Wi-Fi driver can send: `wifi_tx_pkt_size_max`

**So, the peak memory that the Wi-Fi driver consumes can be calculated with the following formula:**

$$\text{wifi\_dynamic\_peek\_memory} = (\text{wifi\_rx\_dynamic\_buf\_num} * \text{wifi\_rx\_pkt\_size\_max}) + (\text{wifi\_tx\_dynamic\_buf\_num} * \text{wifi\_tx\_pkt\_size\_max})$$

Generally, we do not need to care about the dynamic tx long buffers and dynamic tx long long buffers, because they are management frames which only have a small impact on the system.

### 4.15.18 Wi-Fi Menuconfig

## Wi-Fi Buffer Configure

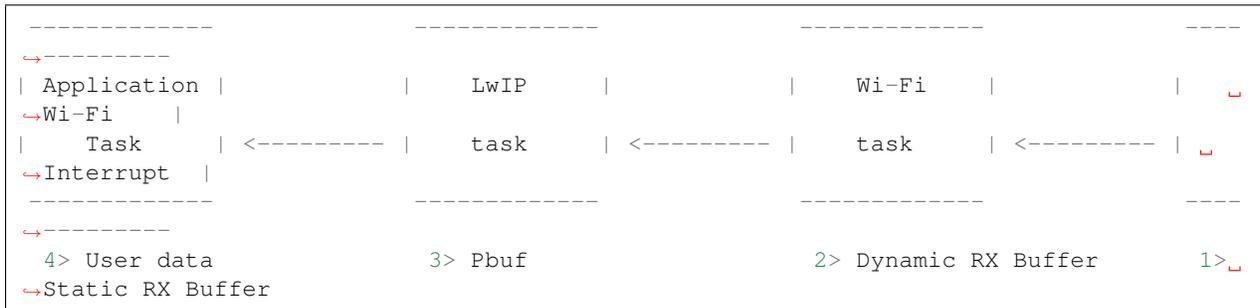
If you are going to modify the default number or type of buffer, it would be helpful to also have an overview of how the buffer is allocated/freed in the data path. The following diagram shows this process in the TX direction:



### Description:

- The application allocates the data which needs to be sent out.
- The application calls TCP/IP-/Socket-related APIs to send the user data. These APIs will allocate a PBUF used in LwIP, and make a copy of the user data.
- When LwIP calls a Wi-Fi API to send the PBUF, the Wi-Fi API will allocate a “Dynamic Tx Buffer” or “Static Tx Buffer”, make a copy of the LwIP PBUF, and finally send the data.

The following diagram shows how buffer is allocated/freed in the RX direction:



### Description:

- The Wi-Fi hardware receives a packet over the air and puts the packet content to the “Static Rx Buffer”, which is also called “RX DMA Buffer”.
- The Wi-Fi driver allocates a “Dynamic Rx Buffer”, makes a copy of the “Static Rx Buffer”, and returns the “Static Rx Buffer” to hardware.
- The Wi-Fi driver delivers the packet to the upper-layer (LwIP), and allocates a PBUF for holding the “Dynamic Rx Buffer”.
- The application receives data from LwIP.

The diagram shows the configuration of the Wi-Fi internal buffer.

Buffer Type	Alloc Type	Default	Configurable	Description
Static RX Buffer (Hardware RX Buffer)	Static	10 * 1600 Bytes	Yes	This is a kind of DMA memory. It is initialized in esp_wifi_init() and freed in esp_wifi_deinit(). 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.
Dynamic RX Buffer	Dynamic	32	Yes	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 menuconfig, is used to limit the total un-freed Dynamic Rx Buffer number.
Dynamic TX Buffer	Dynamic	32	Yes	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
858				Chapter 4. API Guides are mutually exclusive.
Static TX Buffer	Static	32 * 1600Bytes	Yes	This is a kind of DMA

## Wi-Fi NVS Flash

If the Wi-Fi NVS flash is enabled, all Wi-Fi configurations set via the Wi-Fi APIs will be stored into flash, and the Wi-Fi driver will start up with these configurations next time it powers on/reboots. However, the application can choose to disable the Wi-Fi NVS flash if it does not need to store the configurations into persistent memory, or has its own persistent storage, or simply due to debugging reasons, etc.

## Wi-Fi AMPDU

Generally, the AMPDU should be enabled, because it can greatly improve the Wi-Fi throughput. Disabling AMPDU is usually for debugging purposes. It may be removed from future releases.

# 4.16 Support for external RAM

## 4.16.1 Introduction

The ESP32 has a few hundred KiB of internal RAM, residing on the same die as the rest of the ESP32. For some purposes, this is insufficient, and therefore the ESP32 incorporates the ability to also use up to 4MiB of external SPI RAM memory as memory. The external memory is incorporated in the memory map and is, within certain restrictions, usable in the same way internal data RAM is.

## 4.16.2 Hardware

The ESP32 supports SPI (P)SRAM connected in parallel with the SPI flash chip. While the ESP32 is capable of supporting several types of RAM chips, the ESP32 SDK at the moment only supports the ESP-PSRAM32 chip.

The ESP-PSRAM32 chip is an 1.8V device, and can only be used in parallel with an 1.8V flash part. Make sure to either set the MTDI pin to a high signal level on bootup, or program the fuses in the ESP32 to always use a VDD\_SIO level of 1.8V. Not doing this risks damaging the PSRAM and/or flash chip.

**To connect the ESP-PSRAM chip to the ESP32D0W\*, connect the following signals:**

- PSRAM /CE (pin 1) - ESP32 GPIO 16
- PSRAM SO (pin 2) - flash DO
- PSRAM SIO[2] (pin 3) - flash WP
- PSRAM SI (pin 5) - flash DI
- PSRAM SCLK (pin 6) - ESP32 GPIO 17
- PSRAM SIO[3] (pin 7) - flash HOLD
- PSRAM Vcc (pin 8) - ESP32 VCC\_SDIO

Connections for the ESP32D2W\* chips are TBD.

---

**Note:** Espressif sells an ESP-WROVER module which contains an ESP32, 1.8V flash and the ESP-PSRAM32 integrated in a module, ready for inclusion on an end product PCB.

---

### 4.16.3 Software

**ESP-IDF fully supports integrating external memory use into your applications. ESP-IDF can be configured to handle external**

- Only initialize RAM. This allows the application to manually place data here by dereferencing pointers pointed at the external RAM memory region (0x3F800000 and up).
- Initialize RAM and add it to the capability allocator. This allows a program to specifically allocate a chunk of external RAM using `heap_caps_malloc(size, MALLOC_CAP_SPIRAM)`. This memory can be used and subsequently freed using a normal `free()` call.
- Initialize RAM, add it to the capability allocator and add memory to the pool of RAM that can be returned by `malloc()`. This allows any application to use the external RAM without having to rewrite the code to use `heap_caps_malloc`.

All these options can be selected from the menuconfig menu.

#### Restrictions

**The use of external RAM has a few restrictions:**

- When disabling flash cache (for example, because the flash is being written to), the external RAM also becomes inaccessible; any reads from or writes to it will lead to an illegal cache access exception. This is also the reason that ESP-IDF will never allocate a tasks stack in external RAM.
- External RAM cannot be used as a place to store DMA transaction descriptors or as a buffer for a DMA transfer to read from or write into. Any buffers that will be used in combination with DMA must be allocated using `heap_caps_malloc(size, MALLOC_CAP_DMA)` (and can be freed using a standard `free()` call.)
- External RAM uses the same cache region as the external flash. This means that often accessed variables in external RAM can be read and modified almost as quickly as in internal ram. However, when accessing large chunks of data (>32K), 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 execution of code afterwards slower.
- External RAM cannot be used as task stack memory; because of this, `xTaskCreate` and similar functions will always allocate internal memory for stack and task TCBS and `xTaskCreateStatic`-type functions will check if the buffers passed are internal. However, for tasks not calling on code in ROM in any way, directly or indirectly, the menuconfig option `SPIRAM_ALLOW_STACK_EXTERNAL_MEMORY` will eliminate the check in `xTaskCreateStatic`, allowing task stack in external RAM. Using this is not advised, however.

Because there are a fair few situations that have a specific need for internal memory, but it is also possible to use `malloc()` to exhaust internal memory, there is a pool reserved specifically for requests that cannot be resolved from external memory; allocating task stack, DMA buffers and memory that stays accessible when cache is disabled is drawn from this pool. The size of this pool is configurable in menuconfig.

### 4.16.4 Chip revisions

There are some issues with certain revisions of the ESP32 that have repercussions for use with external RAM. These are documented in the ESP32 [ECO](#) document. In particular, ESP-IDF handles the bugs mentioned in the following ways:

### ESP32 rev v0

ESP-IDF has no workaround for the bugs in this revision of silicon, and it cannot be used to map external PSRAM into the ESP32s main memory map.

### ESP32 rev v1

The bugs in this silicon revision introduce a hazard when certain sequences of machine instructions operate on external memory locations (ESP32 ECO 3.2). To work around this, the gcc compiler to compile ESP-IDF has been expanded with a flag: `-mfix-esp32-psram-cache-issue`. With this flag passed to gcc on the command line, the compiler works around these sequences and only outputs code that can safely be executed.

In ESP-IDF, this flag is enabled when you select `SPIRAM_CACHE_WORKAROUND`. ESP-IDF also takes other measures to make sure no combination of PSRAM access plus the offending instruction sets are used: it links to a version of Newlib recompiled with the gcc flag, doesn't use some ROM functions and allocates static memory for the WiFi stack.



We welcome contributions to the esp-idf project!

## 5.1 How to Contribute

Contributions to esp-idf - fixing bugs, adding features, adding documentation - are welcome. We accept contributions via [Github Pull Requests](#).

## 5.2 Before Contributing

Before sending us a Pull Request, please consider this list of points:

- Is the contribution entirely your own work, or already licensed under an Apache License 2.0 compatible Open Source License? If not then we unfortunately cannot accept it.
- Does any new code conform to the esp-idf *Style Guide*?
- Does the code documentation follow requirements in *Documenting Code*?
- Is the code adequately commented for people to understand how it is structured?
- Is there documentation or examples that go with code contributions? There are additional suggestions for writing good examples in [examples](#) readme.
- Are comments and documentation written in clear English, with no spelling or grammar errors?
- If the contribution contains multiple commits, are they grouped together into logical changes (one major change per pull request)? Are any commits with names like “fixed typo” [squashed into previous commits](#)?
- If you’re unsure about any of these points, please open the Pull Request anyhow and then ask us for feedback.

## 5.3 Pull Request Process

After you open the Pull Request, there will probably be some discussion in the comments field of the request itself.

Once the Pull Request is ready to merge, it will first be merged into our internal git system for in-house automated testing.

If this process passes, it will be merged onto the public github repository.

## 5.4 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.

## 5.5 Related Documents

### 5.5.1 Espressif IoT Development Framework Style Guide

#### About this guide

Purpose of this style guide is to encourage use of common coding practices within the ESP-IDF.

Style guide is a set of rules which are aimed to help create readable, maintainable, and robust code. By writing code which looks the same way across the code base we help others read and comprehend the code. By using same conventions for spaces and newlines we reduce chances that future changes will produce huge unreadable diffs. By following common patterns for module structure and by using language features consistently we help others understand code behavior.

We try to keep rules simple enough, which means that they can not cover all potential cases. In some cases one has to bend these simple rules to achieve readability, maintainability, or robustness.

When doing modifications to third-party code used in ESP-IDF, follow the way that particular project is written. That will help propose useful changes for merging into upstream project.

#### C code formatting

##### Indentation

Use 4 spaces for each indentation level. Don't use tabs for indentation. Configure the editor to emit 4 spaces each time you press tab key.

##### Vertical space

Place one empty line between functions. Don't begin or end a function with an empty line.

```
void function1()
{
    do_one_thing();
    do_another_thing();
    // INCORRECT, don't place empty line here
```

(continues on next page)

(continued from previous page)

```

}
                                // place empty line here
void function2()
{
                                // INCORRECT, don't use an empty line here
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
    }
}

```

## Horizontal space

Always add single space after conditional and loop keywords:

```

if (condition) {    // correct
    // ...
}

switch (n) {        // correct
    case 0:
        // ...
}

for(int i = 0; i < CONST; ++i) {    // INCORRECT
    // ...
}

```

Add single space around binary operators. No space is necessary for unary operators. It is okay to drop space around multiply and divide operators:

```

const int y = y0 + (x - x0) * (y1 - y0) / (x1 - x0);    // correct

const int y = y0 + (x - x0)*(y1 - y0)/(x1 - x0);        // also okay

int y_cur = -y;                                          // correct
++y_cur;

const int y = y0+(x-x0)*(y1-y0)/(x1-x0);                // INCORRECT

```

No space is necessary around `.` and `->` operators.

Sometimes adding horizontal space within a line can help make code more readable. For example, you can add space to align function arguments:

```

gpio_matrix_in(PIN_CAM_D6,    I2S0I_DATA_IN14_IDX, false);
gpio_matrix_in(PIN_CAM_D7,    I2S0I_DATA_IN15_IDX, false);
gpio_matrix_in(PIN_CAM_HREF,  I2S0I_H_ENABLE_IDX,  false);
gpio_matrix_in(PIN_CAM_PCLK,  I2S0I_DATA_IN15_IDX, false);

```

Note however that if someone goes to add new line with a longer identifier as first argument (e.g. `PIN_CAM_VSYNC`), it will not fit. So other lines would have to be realigned, adding meaningless changes to the commit.

Therefore, use horizontal alignment sparingly, especially if you expect new lines to be added to the list later.

Never use TAB characters for horizontal alignment.

Never add trailing whitespace at the end of the line.

## Braces

- Function definition should have a brace on a separate line:

```
// This is correct:
void function(int arg)
{

}

// NOT like this:
void function(int arg) {

}
```

- Within a function, place opening brace on the same line with conditional and loop statements:

```
if (condition) {
    do_one();
} else if (other_condition) {
    do_two();
}
```

## Comments

Use `//` for single line comments. For multi-line comments it is okay to use either `//` on each line or a `/* */` block. Although not directly related to formatting, here are a few notes about using comments effectively.

- Don't use single comments to disable some functionality:

```
void init_something()
{
    setup_dma();
    // load_resources(); // WHY is this thing commented, asks the_
↪reader?
    start_timer();
}
```

- If some code is no longer required, remove it completely. If you need it you can always look it up in git history of this file. If you disable some call because of temporary reasons, with an intention to restore it in the future, add explanation on the adjacent line:

```
void init_something()
{
    setup_dma();
    // TODO: we should load resources here, but loader is not fully integrated_
↪yet.
    // load_resources();
    start_timer();
}
```

- Same goes for `#if 0 ... #endif` blocks. Remove code block completely if it is not used. Otherwise, add comment explaining why the block is disabled. Don't use `#if 0 ... #endif` or comments to store code snippets which you may need in the future.
- Don't add trivial comments about authorship and change date. You can always look up who modified any given line using git. E.g. this comment adds clutter to the code without adding any useful information:

```
void init_something()
{
    setup_dma();
    // XXX add 2016-09-01
    init_dma_list();
    fill_dma_item(0);
    // end XXX add
    start_timer();
}
```

## Formatting your code

You can use `astyle` program to format your code according to the above recommendations.

If you are writing a file from scratch, or doing a complete rewrite, feel free to re-format the entire file. If you are changing a small portion of file, don't re-format the code you didn't change. This will help others when they review your changes.

To re-format a file, run:

```
tools/format.sh components/my_component/file.c
```

## Configuring the code style for a project using EditorConfig

EditorConfig helps developers define and maintain consistent coding styles between different editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

For more information, see [EditorConfig Website](#).

## Documenting code

Please see the guide here: *Documenting Code*.

## Structure and naming

### Language features

To be written.

## 5.5.2 Documenting Code

The purpose of this description is to provide quick summary on documentation style used in [espressif/esp-idf](#) repository and how to add new documentation.

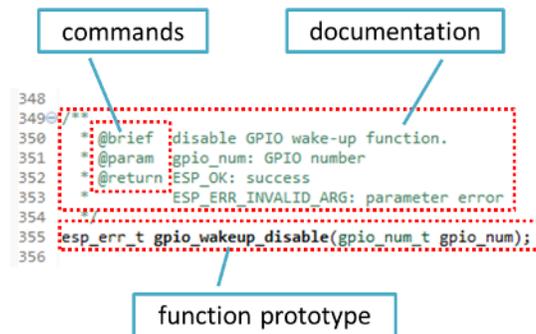
## Introduction

When documenting code for this repository, please follow [Doxygen style](#). You are doing it by inserting special commands, for instance `@param`, into standard comments blocks, for example:

```
/**
 * @param ratio this is oxygen to air ratio
 */
```

Doxygen is phrasing the code, extracting the commands together with subsequent text, and building documentation out of it.

Typical comment block, that contains documentation of a function, looks like below.

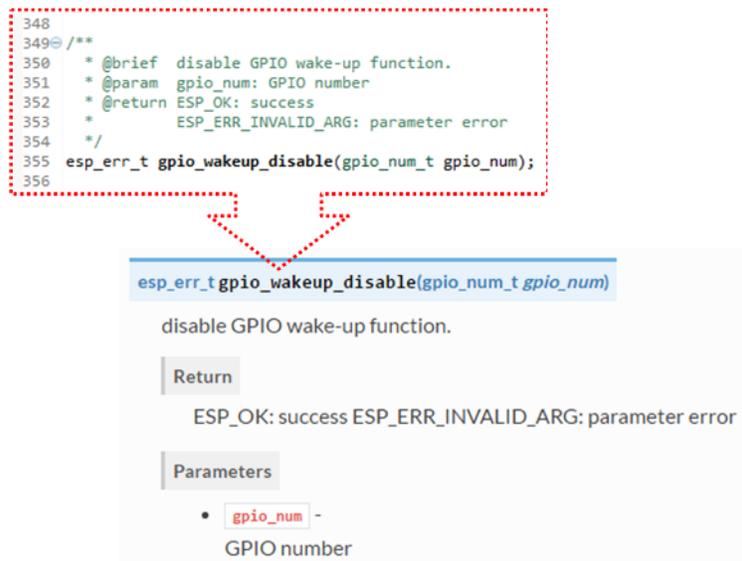


Doxygen supports couple of formatting styles. It also gives you great flexibility on level of details to include in documentation. To get familiar with available features, please check data reach and very well organized [Doxygen Manual](#).

## Why we need it?

The ultimate goal is to ensure that all the code is consistently documented, so we can use tools like [Sphinx](#) and [Breathe](#) to aid preparation and automatic updates of API documentation when the code changes.

With these tools the above piece of code renders like below:



## Go for it!

When writing code for this repository, please follow guidelines below.

1. Document all building blocks of code: functions, structs, typedefs, enums, macros, etc. Provide enough information on purpose, functionality and limitations of documented items, as you would like to see them documented when reading the code by others.
2. Documentation of function should describe what this function does. If it accepts input parameters and returns some value, all of them should be explained.
3. Do not add a data type before parameter or any other characters besides spaces. All spaces and line breaks are compressed into a single space. If you like to break a line, then break it twice.

```

41 @ /**
42  * @brief Set log level for given tag
43  *
44  * If logging for given component has already been enabled, changes previous setting.
45  *
46  * @param tag Tag of the log entries to enable. Must be a non-NULL zero terminated string.
47  *          Value "" resets log level for all tags to the given value.
48  *
49  * @param level Selects log level to enable.
50  *            Only logs at this and lower levels will be shown.
51  */
52 void esp_log_level_set(const char* tag, esp_log_level_t level);

```

do not add data type

white spaces are compressed

a line break that will render this line break will not render

```

void esp_log_level_set(const char*tag, esp_log_level_t level)

```

Set log level for given tag.

If logging for given component has already been enabled, changes previous setting.

Parameters

- **tag** - Tag of the log entries to enable. Must be a non-NULL zero terminated string. Value "" resets log level for all tags to the given value.
- **level** - Selects log level to enable. Only logs at this and lower levels will be shown.

4. If function has void input or does not return any value, then skip @param or @return

```

26 @ /**
27  * @brief Initialize BT controller
28  *
29  * This function should be called only once,
30  * before any other BT functions are called.
31  */
32 void bt_controller_init(void);

```

```

void bt_controller_init(void)

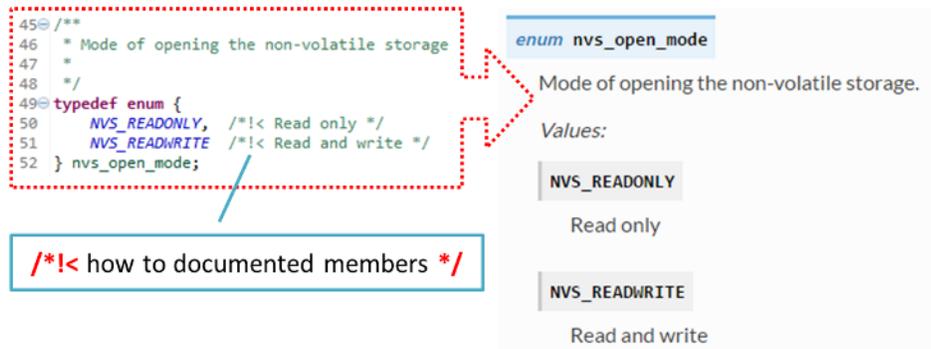
```

Initialize BT controller.

This function should be called only once, before any other BT functions are called.

5. When documenting a define as well as members of a struct or enum, place specific comment like below

after each member.



- To provide well formatted lists, break the line after command (like @return in 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_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
*
    
```

- Overview of functionality of documented header file, or group of files that make a library, should be placed in the same directory in a separate README.rst file. If directory contains header files for different APIs, then the file name should be apiname-readme.rst.

### Go one extra mile

There is couple of tips, how you can make your documentation even better and more useful to the reader.

- Add code snippets to illustrate implementation. To do so, enclose snippet using @code{c} and @endcode commands.

```

*
* @code{c}
* // Example of using nvs_get_i32:
* int32_t max_buffer_size = 4096; // default value
* esp_err_t err = nvs_get_i32(my_handle, "max_buffer_size", &max_buffer_size);
* assert(err == ESP_OK || err == ESP_ERR_NVS_NOT_FOUND);
* // if ESP_ERR_NVS_NOT_FOUND was returned, max_buffer_size will still
* // have its default value.
* @endcode
*
    
```

The code snippet should be enclosed in a comment block of the function that it illustrates.

- 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.

- 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](#).

- You may want to go even further and skip some code like e.g. repetitive defines or enumerations. In such case enclose the code within `/** @cond */` and `/** @endcond */` commands. Example of such implementation is provided in [driver/include/driver/gpio.h](#).
- Use markdown to make your documentation even more readable. You will add headers, links, tables and more.

```
*
* [ESP32 Technical Reference] (https://espressif.com/sites/default/files/
* ↪documentation/esp32\_technical\_reference\_manual\_en.pdf)
*
```

---

**Note:** Code snippets, notes, links, etc. will not make it to the documentation, if not enclosed in a comment block associated with one of documented objects.

---

- Prepare one or more complete code examples together with description. Place description in a separate file `README.md` in specific folder of [examples](#) directory.

## Linking Examples

When linking to examples on GitHub do not use absolute / hardcoded URLs. Instead, use docutils custom roles that will generate links for you. These auto-generated links point to the tree or blob for the git commit ID (or tag) of the repository. This is needed to ensure that links do not get broken when files in master branch are moved around or deleted.

The following roles are provided:

- `:idf:`path`` - points to directory inside ESP-IDF
- `:idf_file:`path`` - points to file inside ESP-IDF
- `:idf_raw:`path`` - points to raw view of the file inside ESP-IDF
- `:component:`path`` - points to directory inside ESP-IDF components dir
- `:component_file:`path`` - points to file inside ESP-IDF components dir
- `:component_raw:`path`` - points to raw view of the file inside ESP-IDF components dir
- `:example:`path`` - points to directory inside ESP-IDF examples dir
- `:example_file:`path`` - points to file inside ESP-IDF examples dir
- `:example_raw:`path`` - points to raw view of the file inside ESP-IDF examples dir

A check is added to the CI build script, which searches RST files for presence of hard-coded links (identified by `tree/master`, `blob/master`, or `raw/master` part of the URL). This check can be run manually: `cd docs` and then `make gh-linkcheck`.

### Add Illustrations

Consider adding diagrams and pictures to illustrate described concepts.

Sometimes it is better to add an illustration than writing a lengthy paragraph to describe a complex idea, a data structure or an algorithm. This repository is using `blockdiag` suite of tools to generate diagram images from simple text files.

The following types of diagrams are supported:

- [Block diagram](#)
- [Sequence diagram](#)
- [Activity diagram](#)
- [Logical network diagram](#)

With this suite of tools it is possible to generate beautiful diagram images from simple text format (similar to `graphviz`'s DOT format). The diagram elements are laid out automatically. The diagram code is then converted into “.png” graphics and integrated “behind the scenes” into **Sphinx** documents.

For the diagram preparation you can use an on-line [interactive shell](#) that instantly shows the rendered image.

Below are couple of diagram examples:

- Simple **block diagram** / [blockdiag - Wi-Fi Buffer Configuration](#)
- Slightly more complicated **block diagram** - [Wi-Fi programming model](#)
- **Sequence diagram** / [seqdiag - Scan for a Specific AP in All Channels](#)
- **Packet diagram** / [packetdiag - NVS Page Structure](#)

Try them out by modifying the source code and see the diagram instantly rendering below.

---

**Note:** There may be slight differences in rendering of font used by the [interactive shell](#) compared to the font used in the `esp-idf` documentation.

---

### Put it all together

Once documentation is ready, follow instruction in *API Documentation Template* and create a single file, that will merge all individual pieces of prepared documentation. Finally add a link to this file to respective `.. toctree::` in `index.rst` file located in `/docs` folder or subfolders.

### OK, but I am new to Sphinx!

1. No worries. All the software you need is well documented. It is also open source and free. Start by checking [Sphinx](#) documentation. If you are not clear how to write using rst markup language, see [reStructuredText Primer](#).
2. Check the source files of this documentation to understand what is behind of what you see now on the screen. Sources are maintained on GitHub in [espressif/esp-idf](#) repository in `docs` folder. You can go directly to the source file of this page by scrolling up and clicking the link in the top right corner. When on GitHub, see what's really inside, open source files by clicking `Raw` button.

3. You will likely want to see how documentation builds and looks like before posting it on the GitHub. There are two options to do so:
  - Install [Sphinx](#), [Breathe](#), [Blockdiag](#) and [Doxygen](#) to build it locally, see chapter below.
  - Set up an account on [Read the Docs](#) and build documentation in the cloud. Read the Docs provides document building and hosting for free and their service works really quick and great.
4. To preview documentation before building, use [Sublime Text](#) editor together with [OmniMarkupPreviewer](#) plugin.

## Setup for building documentation locally

You can setup environment to build documentation locally on your PC by installing:

1. Doxygen - <https://www.stack.nl/~dimitri/doxygen/>
2. Sphinx - <https://github.com/sphinx-doc/sphinx/#readme-for-sphinx>
3. Document theme “sphinx\_rtd\_theme” - [https://github.com/rtfd/sphinx\\_rtd\\_theme](https://github.com/rtfd/sphinx_rtd_theme)
4. Breathe - <https://github.com/michaeljones/breathe#breathe>
5. Blockdiag - <http://blockdiag.com/en/index.html>

The package “sphinx\_rtd\_theme” is added to have the same “look and feel” of [ESP32 Programming Guide](#) documentation like on the “Read the Docs” hosting site.

Do not worry about being confronted with several packages to install. Besides Doxygen, all remaining packages are written in Python. Therefore installation of all of them is combined into one simple step.

Installation of Doxygen is OS dependent:

### Linux

```
sudo apt-get install doxygen
```

**Windows** - install in MSYS2 console

```
pacman -S doxygen
```

### MacOS

```
brew install doxygen
```

---

**Note:** If you are installing on Windows system (Linux and MacOS users should skip this note), **before** going further, execute two extra steps below. These steps are required for the [blockdiag](#) to install:

1. Update all the system packages:

```
$ pacman -Syu
```

This process will likely require restarting of the MSYS2 MINGW32 console and repeating above commands, until update is complete.

2. Install [pillow](#), that is one of dependences of the [blockdiag](#):

```
$ pacman -S mingw32/mingw-w64-i686-python2-pillow
```

Check the log on the screen that `mingw-w64-i686-python2-pillow-4.3.0-1` is installed. Previous versions of [pillow](#) will not work.

A downside of Windows installation is that fonts of the *blockdiag pictures* <add-illustrations> do not render correctly, you will see some random characters instead. Until this issue is fixed, you can use the [interactive shell](#) to see how the complete picture looks like.

---

All remaining applications are [Python](#) packages and you can install them in one step as follows:

```
cd ~/esp/esp-idf/docs
pip install -r requirements.txt
```

---

**Note:** Installation steps assume that ESP-IDF is placed in `~/esp/esp-idf` directory, that is default location of ESP-IDF used in documentation.

---

Now you should be ready to build documentation by invoking:

```
make html
```

---

This may take couple of minutes. After completion, documentation will be placed in `~/esp/esp-idf/docs/_build/html` folder. To see it, open `index.html` in a web browser.

### Wrap up

We love good code that is doing cool things. We love it even better, if it is well documented, so we can quickly make it run and also do the cool things.

Go ahead, contribute your code and documentation!

### Related Documents

- [API Documentation Template](#)

## 5.5.3 API Documentation Template

---

**Note:** *INSTRUCTIONS*

1. Use this file (`docs/api-reference/template.rst`) as a template to document API.
  2. Change the file name to the name of the header file that represents documented API.
  3. Include respective files with descriptions from the API folder using `..include::`
    - `README.rst`
    - `example.rst`
    - ...
  4. Optionally provide description right in this file.
  5. Once done, remove all instructions like this one and any superfluous headers.
-

## Overview

---

**Note:** *INSTRUCTIONS*

1. Provide overview where and how this API may be used.
  2. Where applicable include code snippets to illustrate functionality of particular functions.
  3. To distinguish between sections, use the following [heading levels](#):
    - # with overline, for parts
    - \* with overline, for chapters
    - =, for sections
    - -, for subsections
    - ^, for subsubsections
    - ", for paragraphs
- 

## Application Example

---

**Note:** *INSTRUCTIONS*

1. Prepare one or more practical examples to demonstrate functionality of this API.
  2. Each example should follow pattern of projects located in `esp-idf/examples/` folder.
  3. Place example in this folder complete with `README.md` file.
  4. Provide overview of demonstrated functionality in `README.md`.
  5. With good overview reader should be able to understand what example does without opening the source code.
  6. Depending on complexity of example, break down description of code into parts and provide overview of functionality of each part.
  7. Include flow diagram and screenshots of application output if applicable.
  8. Finally add in this section synopsis of each example together with link to respective folder in `esp-idf/examples/`.
- 

## API Reference

---

**Note:** *INSTRUCTIONS*

1. This repository provides for automatic update of API reference documentation using *code markup retrieved by Doxygen from header files*.
2. Update is done on each documentation build by invoking script `docs/gen-dxd.py` for all header files listed in the `INPUT` statement of `docs/Doxyfile`.
3. Each line of the `INPUT` statement (other than a comment that begins with `##`) contains a path to header file `*.h` that will be used to generate corresponding `*.inc` files:

```
##
## Wi-Fi - API Reference
##
../components/esp32/include/esp_wifi.h \
../components/esp32/include/esp_smartconfig.h \
```

4. The \*.inc files contain formatted reference of API members generated automatically on each documentation build. All \*.inc files are placed in Sphinx \_build directory. To see directives generated for e.g. esp\_wifi.h, run `python gen-dxd.py esp32/include/esp_wifi.h`.
5. To show contents of \*.inc file in documentation, include it as follows:

```
.. include:: /_build/inc/esp_wifi.inc
```

For example see [docs/api-reference/wifi/esp\\_wifi.rst](#)

6. Optionally, rather than using \*.inc files, you may want to describe API in your own way. See [docs/api-reference/system/deep\\_sleep.rst](#) for example.

Below is the list of common .. doxygen...:: directives:

- Functions - .. doxygenfunction:: name\_of\_function
- Unions - .. doxygenunion:: name\_of\_union
- Structures - .. doxygenstruct:: name\_of\_structure together with :members:
- Macros - .. doxygendefine:: name\_of\_define
- Type Definitions - .. doxygentypedef:: name\_of\_type
- Enumerations - .. doxygenenum:: name\_of\_enumeration

See [Breathe documentation](#) for additional information.

To provide a link to header file, use the *link custom role* as follows:

```
* :component_file:`path_to/header_file.h`
```

7. In any case, to generate API reference, the file [docs/Doxyfile](#) should be updated with paths to \*.h headers that are being documented.
  8. When changes are committed and documentation is built, check how this section has been rendered. *Correct annotations* in respective header files, if required.
- 

## 5.5.4 Contributor Agreement

### Individual Contributor Non-Exclusive License Agreement

#### including the Traditional Patent License OPTION

Thank you for your interest in contributing to Espressif IoT Development Framework (esp-idf) (“We” or “Us”).

The purpose of this contributor agreement (“Agreement”) is to clarify and document the rights granted by contributors to Us. To make this document effective, please follow the instructions at [CONTRIBUTING.rst](#)

## 1. DEFINITIONS

“**You**” means the Individual Copyright owner who submits a Contribution to Us. If You are an employee and submit the Contribution as part of your employment, You have had Your employer approve this Agreement or sign the Entity version of this document.

“**Contribution**” means any original work of authorship (software and/or documentation) including any modifications or additions to an existing work, Submitted by You to Us, in which You own the Copyright. If You do not own the Copyright in the entire work of authorship, please contact Us at [angus@espressif.com](mailto:angus@espressif.com).

“**Copyright**” means all rights protecting works of authorship owned or controlled by You, including copyright, moral and neighboring rights, as appropriate, for the full term of their existence including any extensions by You.

“**Material**” means the software or documentation made available by Us to third parties. When this Agreement covers more than one software project, the Material means the software or documentation to which the Contribution was Submitted. After You Submit the Contribution, it may be included in the Material.

“**Submit**” means any form of physical, electronic, or written communication sent to Us, including but not limited to electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, Us, but excluding communication that is conspicuously marked or otherwise designated in writing by You as “Not a Contribution.”

“**Submission Date**” means the date You Submit a Contribution to Us.

“**Documentation**” means any non-software portion of a Contribution.

## 2. LICENSE GRANT

### 2.1 Copyright License to Us

Subject to the terms and conditions of this Agreement, You hereby grant to Us a worldwide, royalty-free, NON-exclusive, perpetual and irrevocable license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, under the Copyright covering the Contribution to use the Contribution by all means, including, but not limited to:

- to publish the Contribution,
- to modify the Contribution, to prepare derivative works based upon or containing the Contribution and to combine the Contribution with other software code,
- to reproduce the Contribution in original or modified form,
- to distribute, to make the Contribution available to the public, display and publicly perform the Contribution in original or modified form.

2.2 Moral Rights remain unaffected to the extent they are recognized and not waivable by applicable law. Notwithstanding, You may add your name in the header of the source code files of Your Contribution and We will respect this attribution when using Your Contribution.

## 3. PATENTS

### 3.1 Patent License

Subject to the terms and conditions of this Agreement You hereby grant to us a worldwide, royalty-free, non-exclusive, perpetual and irrevocable (except as stated in Section 3.2) patent license, with the right to transfer an unlimited number of non-exclusive licenses or to grant sublicenses to third parties, to make, have made, use, sell, offer for sale, import and otherwise transfer the Contribution and the Contribution in combination with the Material (and portions of such combination). This license applies to all patents owned or controlled by You, whether already acquired or hereafter

acquired, that would be infringed by making, having made, using, selling, offering for sale, importing or otherwise transferring of Your Contribution(s) alone or by combination of Your Contribution(s) with the Material.

### 3.2 Revocation of Patent License

You reserve the right to revoke the patent license stated in section 3.1 if we make any infringement claim that is targeted at your Contribution and not asserted for a Defensive Purpose. An assertion of claims of the Patents shall be considered for a “Defensive Purpose” if the claims are asserted against an entity that has filed, maintained, threatened, or voluntarily participated in a patent infringement lawsuit against Us or any of Our licensees.

## 4. DISCLAIMER

THE CONTRIBUTION IS PROVIDED “AS IS”. MORE PARTICULARLY, ALL EXPRESS OR IMPLIED WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED BY YOU TO US AND BY US TO YOU. TO THE EXTENT THAT ANY SUCH WARRANTIES CANNOT BE DISCLAIMED, SUCH WARRANTY IS LIMITED IN DURATION TO THE MINIMUM PERIOD PERMITTED BY LAW.

## 5. Consequential Damage Waiver

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL YOU OR US BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF ANTICIPATED SAVINGS, LOSS OF DATA, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL AND EXEMPLARY DAMAGES ARISING OUT OF THIS AGREEMENT REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED.

## 6. Approximation of Disclaimer and Damage Waiver

IF THE DISCLAIMER AND DAMAGE WAIVER MENTIONED IN SECTION 4 AND SECTION 5 CANNOT BE GIVEN LEGAL EFFECT UNDER APPLICABLE LOCAL LAW, REVIEWING COURTS SHALL APPLY LOCAL LAW THAT MOST CLOSELY APPROXIMATES AN ABSOLUTE WAIVER OF ALL CIVIL LIABILITY IN CONNECTION WITH THE CONTRIBUTION.

## 7. Term

7.1 This Agreement shall come into effect upon Your acceptance of the terms and conditions.

7.2 In the event of a termination of this Agreement Sections 4, 5, 6, 7 and 8 shall survive such termination and shall remain in full force thereafter. For the avoidance of doubt, Contributions that are already licensed under a free and open source license at the date of the termination shall remain in full force after the termination of this Agreement.

## 8. Miscellaneous

8.1 This Agreement and all disputes, claims, actions, suits or other proceedings arising out of this agreement or relating in any way to it shall be governed by the laws of People’s Republic of China excluding its private international law provisions.

8.2 This Agreement sets out the entire agreement between You and Us for Your Contributions to Us and overrides all other agreements or understandings.

8.3 If any provision of this Agreement is found void and unenforceable, such provision will be replaced to the extent possible with a provision that comes closest to the meaning of the original provision and that is enforceable. The terms and conditions set forth in this Agreement shall apply notwithstanding any failure of essential purpose of this Agreement or any limited remedy to the maximum extent possible under law.

8.4 You agree to notify Us of any facts or circumstances of which you become aware that would make this Agreement inaccurate in any respect.

**You**

Date:		
Name:		
Title:		
Address:		

**Us**

Date:		
Name:		
Title:		
Address:		



The ESP-IDF GitHub repository is updated regularly, especially on the “master branch” where new development happens. There are also stable releases which are recommended for production use.

### 6.1 Releases

Documentation for the current stable version can always be found at this URL:

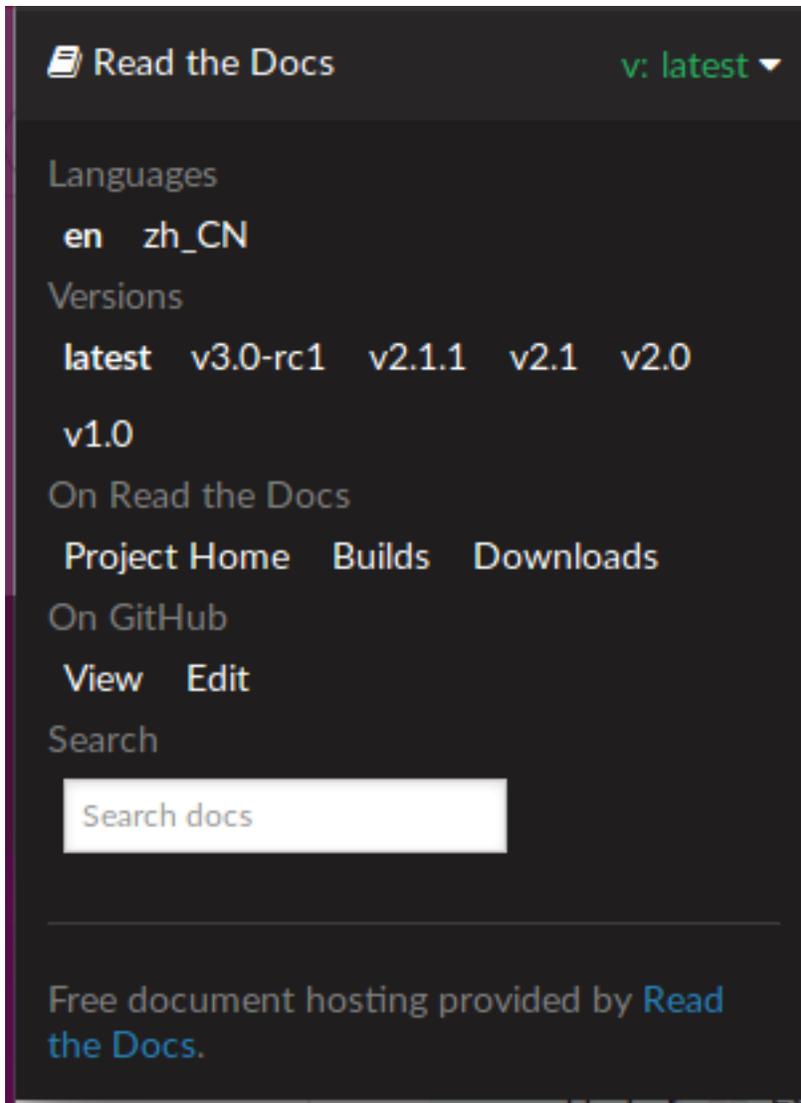
<https://docs.espressif.com/projects/esp-idf/en/stable/>

Documentation for the latest version (“master branch”) can always be found at this URL:

<https://docs.espressif.com/projects/esp-idf/en/latest/>

The full history of releases can be found on the GitHub repository [Releases page](#). There you can find release notes, links to each version of the documentation, and instructions for obtaining each version.

Documentation for all releases can also be found in the HTML documentation by clicking the “versions” pop up in the bottom-left corner of the page. You can use this popup to switch between versions of the documentation.



## 6.2 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).
- 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 don’t want to use the master branch, it is possible to check out a pre-release version or a release branch. It is recommended to start from a stable version and then follow the instructions for [Updating to a Pre-Release Version](#) or [Updating to a Release Branch](#).

See [Updating ESP-IDF](#) if you already have a local copy of ESP-IDF and wish to update it.

## 6.3 Versioning Scheme

ESP-IDF uses [Semantic Versioning](#). This means:

- Major Releases like `v3.0` add new functionality and may change functionality. This includes removing deprecated functionality.

When 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 will 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`) then none of your project's code should need updating, but you should re-test your project. Pay particular attention to 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 should not need to change any code in your project and should only need to re-test functionality relating directly to bugs listed in the release notes on the [Releases page](#).

## 6.4 Checking The Current Version

The local ESP-IDF version can be checked using git:

```
cd $IDF_PATH
git describe --tags --dirty
```

The 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 (these versions in code will not always update, it only changes if that particular source file is recompiled).

Examples of ESP-IDF versions:

Version String	Meaning
<code>v3.2-dev-306-gbeb3611ca</code>	Master branch pre-release, in development for version 3.2. 306 commits after v3.2 development started. Commit identifier <code>beb3611ca</code> .
<code>v3.0.2</code>	Stable release, tagged <code>v3.0.2</code> .
<code>v3.1-beta1-75-g346d6b0ea</code>	Beta version in development (on a <i>release branch</i> ). 75 commits after v3.1-beta1 pre-release tag. Commit identifier <code>346d6b0ea</code> .
<code>v3.0.1-dirty</code>	Stable release, tagged <code>v3.0.1</code> . There are modifications in the local ESP-IDF directory ("dirty").

## 6.5 Git Workflow

The development (Git) workflow of the Espressif ESP-IDF team is:

- 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](#).

## 6.6 Updating ESP-IDF

Updating ESP-IDF depends on which version(s) you wish to follow:

- *Updating to Stable Release* is recommended for production use.
- *Updating to Master Branch* is recommended for latest features, development use, and testing.
- *Updating to a Release Branch* is a compromise between these two.

---

**Note:** These guides assume you already have a local copy of ESP-IDF. To get one, follow the [Getting Started](#) guide for any ESP-IDF version.

---

### 6.6.1 Updating to Stable Release

To update to new ESP-IDF releases (recommended for production use), this is the process to follow:

- Check the [Releases page](#) regularly for new releases.
- When a bugfix release for a version you are using is released (for example if using `v3.0.1` and `v3.0.2` is available), check out the new bugfix version into the existing ESP-IDF directory:

```
cd $IDF_PATH
git fetch
git checkout vX.Y.Z
git submodule update --init --recursive
```

- When major or minor updates are released, check the Release Notes on the releases page and decide if you would like to update or to stay with your existing release. Updating is via the same Git commands shown above.

---

**Note:** If you installed the stable release via zip file rather than using git, it may not be possible to change versions this way. In this case, update by downloading a new zip file and replacing the entire `IDF_PATH` directory with its contents.

---

## 6.6.2 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*.

## 6.6.3 Updating to Master Branch

---

**Note:** Using Master branch means living “on the bleeding edge” with the latest ESP-IDF code.

---

To use the latest version on the ESP-IDF master branch, this is the process to follow:

- Check out the master branch locally:

```
cd $IDF_PATH
git checkout master
git pull
git submodule update --init --recursive
```

- Periodically, re-run `git pull` to pull the latest version of master. Note that you may need to change your project or report bugs after updating master branch.
- To switch from `master` to a release branch or stable version, run `git checkout` as shown in the other sections.

---

**Important:** It is strongly recommended to regularly run `git pull` and then `git submodule update --init --recursive` so a local copy of `master` does not get too old. Arbitrary old master branch revisions are effectively unsupported “snapshots” that may have undocumented bugs. For a semi-stable version, try *Updating to a Release Branch* instead.

---

## 6.6.4 Updating to a Release Branch

In stability terms, using a release branch is part-way between using `master` branch and only using stable releases. A release branch is always beta quality or better, and receives bug fixes before they appear in each stable release.

You can find a [list of branches](#) on GitHub.

For example, to follow the branch for ESP-IDF v3.1, including any bugfixes for future releases like v3.1.1, etc:

```
cd $IDF_PATH
git fetch
git checkout release/v3.1
git pull
git submodule --update --init --recursive
```

Each time you `git pull` this branch, ESP-IDF will be updated with fixes for this release.

---

**Note:** There is no dedicated documentation for release branches. It is recommended to use the documentation for the closest version to the branch which is currently checked out.

---



# CHAPTER 7

---

## Resources

---

- The [esp32.com forum](#) is a place to ask questions and find community resources.
- Check the [Issues](#) section on GitHub if you find a bug or have a feature request. Please check existing [Issues](#) before opening a new one.
- If you're interested in contributing to ESP-IDF, please check the [Contributions Guide](#).
- To develop applications using Arduino platform, refer to [Arduino core for ESP32 WiFi chip](#).
- For additional ESP32 product related information, please refer to [documentation](#) section of [Espressif](#) site.
- Mirror of this documentation is available under: <https://dl.espressif.com/doc/esp-idf/latest/>.



## 8.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2016 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:

- **Newlib** (components/newlib) is licensed under the BSD License and is Copyright of various parties, as described in the file components/newlib/COPYING.NEWLIB.
- Xtensa header files (components/esp32/include/xtensa) are Copyright (C) 2013 Tensilica Inc and are licensed under the MIT License as reproduce in the individual header files.
- **esptool.py** (components/esptool\_py/esptool) is Copyright (C) 2014-2016 Fredrik Ahlberg, Angus Gratton and is licensed under the GNU General Public License v2, as described in the file components/esptool\_py/LICENSE.
- Original parts of **FreeRTOS** (components/freertos) are Copyright (C) 2015 Real Time Engineers Ltd and is licensed under the GNU General Public License V2 with the FreeRTOS Linking Exception, as described in the file components/freertos/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 the file components/lwip/COPYING.
- **KConfig** (tools/kconfig) is Copyright (C) 2002 Roman Zippel and others, and is licensed under the GNU General Public License V2.
- **wpa\_supplicant** Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- **FreeBSD net80211** Copyright (c) 2004-2008 Sam Leffler, Errno Consulting and licensed under the BSD license.
- **JSMN** JSON Parser (components/jsmn) Copyright (c) 2010 Serge A. Zaitsev and licensed under the MIT license.
- **argtable3** argument parsing library Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann and licensed under 3-clause BSD license.
- **linenoise** line editing library Copyright (c) 2010-2014, Salvatore Sanfilippo <antirez at gmail dot com>, Copyright (c) 2010-2013, Pieter Noordhuis <pnoordhuis at gmail dot com>, licensed under 2-clause BSD license.

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

## 8.2 ROM Source Code Copyrights

ESP32 mask ROM hardware includes binaries compiled from portions of the following third party software:

- **Newlib**, as licensed under the BSD License and Copyright of various parties, as described in the file components/newlib/COPYING.NEWLIB.
- **Xtensa libhal**, Copyright (c) Tensilica Inc and licensed under the MIT license (see below).
- **TinyBasic Plus**, Copyright Mike Field & Scott Lawrence and licensed under the MIT license (see below).
- **miniz**, by Rich Geldreich - placed into the public domain.
- **wpa\_supplicant** Copyright (c) 2003-2005 Jouni Malinen and licensed under the BSD license.
- **TJpgDec** Copyright (C) 2011, ChaN, all right reserved. See below for license.

## 8.3 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.

## 8.4 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.

## 8.5 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.



This is documentation of **ESP-IDF**, the framework to develop applications for **ESP32** chip by **Espressif**.  
The **ESP32** is 2.4 GHz Wi-Fi and Bluetooth combo, 32 bit dual core chip with 600 DMIPS processing power.

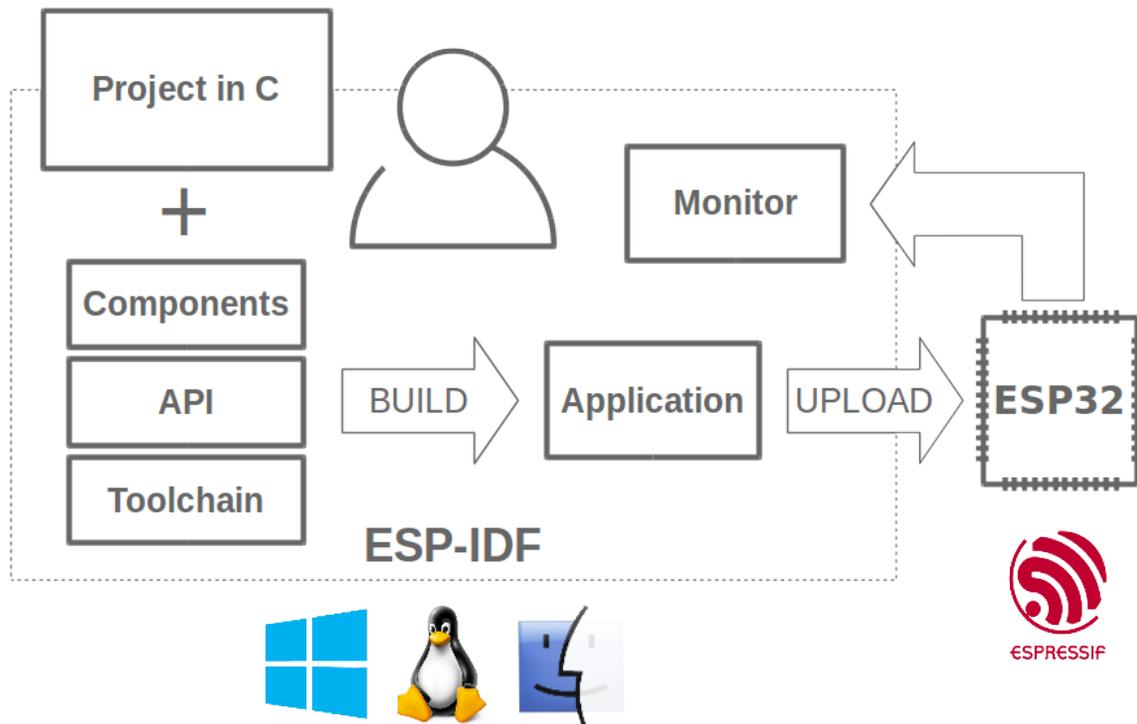


Fig. 1: Espressif IoT Integrated Development Framework

The ESP-IDF, Espressif IoT Integrated Development Framework, provides toolchain, API, components and workflows to develop applications for ESP32 using Windows, Linux and Mac OS operating systems.

- `genindex`

## A

- ADC1\_CHANNEL\_0 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_0\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_1 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_1\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_2 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_2\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_3 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_3\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_4 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_4\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_5 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_5\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_6 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_6\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_7 (*C++ enumerator*), 211
- ADC1\_CHANNEL\_7\_GPIO\_NUM (*C macro*), 216
- ADC1\_CHANNEL\_MAX (*C++ enumerator*), 211
- adc1\_channel\_t (*C++ type*), 211
- adc1\_config\_channel\_atten (*C++ function*), 206
- adc1\_config\_width (*C++ function*), 206
- adc1\_get\_raw (*C++ function*), 207
- ADC1\_GPIO32\_CHANNEL (*C macro*), 216
- ADC1\_GPIO33\_CHANNEL (*C macro*), 216
- ADC1\_GPIO34\_CHANNEL (*C macro*), 216
- ADC1\_GPIO35\_CHANNEL (*C macro*), 216
- ADC1\_GPIO36\_CHANNEL (*C macro*), 216
- ADC1\_GPIO37\_CHANNEL (*C macro*), 216
- ADC1\_GPIO38\_CHANNEL (*C macro*), 216
- ADC1\_GPIO39\_CHANNEL (*C macro*), 216
- adc1\_pad\_get\_io\_num (*C++ function*), 206
- adc1\_ulp\_enable (*C++ function*), 208
- ADC2\_CHANNEL\_0 (*C++ enumerator*), 211
- ADC2\_CHANNEL\_0\_GPIO\_NUM (*C macro*), 216
- ADC2\_CHANNEL\_1 (*C++ enumerator*), 211
- ADC2\_CHANNEL\_1\_GPIO\_NUM (*C macro*), 216
- ADC2\_CHANNEL\_2 (*C++ enumerator*), 211
- ADC2\_CHANNEL\_2\_GPIO\_NUM (*C macro*), 216
- ADC2\_CHANNEL\_3 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_3\_GPIO\_NUM (*C macro*), 216
- ADC2\_CHANNEL\_4 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_4\_GPIO\_NUM (*C macro*), 216
- ADC2\_CHANNEL\_5 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_5\_GPIO\_NUM (*C macro*), 216
- ADC2\_CHANNEL\_6 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_6\_GPIO\_NUM (*C macro*), 217
- ADC2\_CHANNEL\_7 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_7\_GPIO\_NUM (*C macro*), 217
- ADC2\_CHANNEL\_8 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_8\_GPIO\_NUM (*C macro*), 217
- ADC2\_CHANNEL\_9 (*C++ enumerator*), 212
- ADC2\_CHANNEL\_9\_GPIO\_NUM (*C macro*), 217
- ADC2\_CHANNEL\_MAX (*C++ enumerator*), 212
- adc2\_channel\_t (*C++ type*), 211
- adc2\_config\_channel\_atten (*C++ function*), 209
- adc2\_get\_raw (*C++ function*), 209
- ADC2\_GPIO0\_CHANNEL (*C macro*), 216
- ADC2\_GPIO12\_CHANNEL (*C macro*), 216
- ADC2\_GPIO13\_CHANNEL (*C macro*), 216
- ADC2\_GPIO14\_CHANNEL (*C macro*), 217
- ADC2\_GPIO15\_CHANNEL (*C macro*), 216
- ADC2\_GPIO25\_CHANNEL (*C macro*), 217
- ADC2\_GPIO26\_CHANNEL (*C macro*), 217
- ADC2\_GPIO27\_CHANNEL (*C macro*), 217
- ADC2\_GPIO2\_CHANNEL (*C macro*), 216
- ADC2\_GPIO4\_CHANNEL (*C macro*), 216
- adc2\_pad\_get\_io\_num (*C++ function*), 209
- adc2\_vref\_to\_gpio (*C++ function*), 210
- ADC\_ATTEN\_0db (*C macro*), 210
- ADC\_ATTEN\_11db (*C macro*), 210
- ADC\_ATTEN\_2\_5db (*C macro*), 210
- ADC\_ATTEN\_6db (*C macro*), 210
- ADC\_ATTEN\_DB\_0 (*C++ enumerator*), 210
- ADC\_ATTEN\_DB\_11 (*C++ enumerator*), 211
- ADC\_ATTEN\_DB\_2\_5 (*C++ enumerator*), 210
- ADC\_ATTEN\_DB\_6 (*C++ enumerator*), 211
- ADC\_ATTEN\_MAX (*C++ enumerator*), 211

- adc\_atten\_t (C++ type), 210
  - adc\_bits\_width\_t (C++ type), 211
  - ADC\_CHANNEL\_0 (C++ enumerator), 212
  - ADC\_CHANNEL\_1 (C++ enumerator), 212
  - ADC\_CHANNEL\_2 (C++ enumerator), 212
  - ADC\_CHANNEL\_3 (C++ enumerator), 212
  - ADC\_CHANNEL\_4 (C++ enumerator), 212
  - ADC\_CHANNEL\_5 (C++ enumerator), 212
  - ADC\_CHANNEL\_6 (C++ enumerator), 212
  - ADC\_CHANNEL\_7 (C++ enumerator), 212
  - ADC\_CHANNEL\_8 (C++ enumerator), 212
  - ADC\_CHANNEL\_9 (C++ enumerator), 212
  - ADC\_CHANNEL\_MAX (C++ enumerator), 212
  - adc\_channel\_t (C++ type), 212
  - ADC\_ENCODE\_11BIT (C++ enumerator), 213
  - ADC\_ENCODE\_12BIT (C++ enumerator), 213
  - ADC\_ENCODE\_MAX (C++ enumerator), 213
  - adc\_gpio\_init (C++ function), 207
  - ADC\_I2S\_DATA\_SRC\_ADC (C++ enumerator), 213
  - ADC\_I2S\_DATA\_SRC\_IO\_SIG (C++ enumerator), 213
  - ADC\_I2S\_DATA\_SRC\_MAX (C++ enumerator), 213
  - adc\_i2s\_encode\_t (C++ type), 213
  - adc\_i2s\_mode\_init (C++ function), 208
  - adc\_i2s\_source\_t (C++ type), 213
  - adc\_power\_off (C++ function), 207
  - adc\_power\_on (C++ function), 207
  - adc\_set\_clk\_div (C++ function), 208
  - adc\_set\_data\_inv (C++ function), 207
  - adc\_set\_data\_width (C++ function), 206
  - adc\_set\_i2s\_data\_source (C++ function), 208
  - ADC\_UNIT\_1 (C++ enumerator), 212
  - ADC\_UNIT\_2 (C++ enumerator), 213
  - ADC\_UNIT\_ALTER (C++ enumerator), 213
  - ADC\_UNIT\_BOTH (C++ enumerator), 213
  - ADC\_UNIT\_MAX (C++ enumerator), 213
  - adc\_unit\_t (C++ type), 212
  - ADC\_WIDTH\_10Bit (C macro), 210
  - ADC\_WIDTH\_11Bit (C macro), 210
  - ADC\_WIDTH\_12Bit (C macro), 210
  - ADC\_WIDTH\_9Bit (C macro), 210
  - ADC\_WIDTH\_BIT\_10 (C++ enumerator), 211
  - ADC\_WIDTH\_BIT\_11 (C++ enumerator), 211
  - ADC\_WIDTH\_BIT\_12 (C++ enumerator), 211
  - ADC\_WIDTH\_BIT\_9 (C++ enumerator), 211
  - ADC\_WIDTH\_MAX (C++ enumerator), 211
  - ADV\_CHNL\_37 (C++ enumerator), 126
  - ADV\_CHNL\_38 (C++ enumerator), 126
  - ADV\_CHNL\_39 (C++ enumerator), 126
  - ADV\_CHNL\_ALL (C++ enumerator), 126
  - ADV\_FILTER\_ALLOW\_SCAN\_ANY\_CON\_ANY (C++ enumerator), 126
  - ADV\_FILTER\_ALLOW\_SCAN\_ANY\_CON\_WLST (C++ enumerator), 126
  - ADV\_FILTER\_ALLOW\_SCAN\_WLST\_CON\_ANY (C++ enumerator), 126
  - ADV\_FILTER\_ALLOW\_SCAN\_WLST\_CON\_WLST (C++ enumerator), 126
  - ADV\_TYPE\_DIRECT\_IND\_HIGH (C++ enumerator), 126
  - ADV\_TYPE\_DIRECT\_IND\_LOW (C++ enumerator), 126
  - ADV\_TYPE\_IND (C++ enumerator), 126
  - ADV\_TYPE\_NONCONN\_IND (C++ enumerator), 126
  - ADV\_TYPE\_SCAN\_IND (C++ enumerator), 126
- ## B
- BLE\_ADDR\_TYPE\_PUBLIC (C++ enumerator), 102
  - BLE\_ADDR\_TYPE\_RANDOM (C++ enumerator), 102
  - BLE\_ADDR\_TYPE\_RPA\_PUBLIC (C++ enumerator), 102
  - BLE\_ADDR\_TYPE\_RPA\_RANDOM (C++ enumerator), 102
  - BLE\_SCAN\_DUPLICATE\_DISABLE (C++ enumerator), 127
  - BLE\_SCAN\_DUPLICATE\_ENABLE (C++ enumerator), 127
  - BLE\_SCAN\_DUPLICATE\_MAX (C++ enumerator), 127
  - BLE\_SCAN\_FILTER\_ALLOW\_ALL (C++ enumerator), 127
  - BLE\_SCAN\_FILTER\_ALLOW\_ONLY\_WLST (C++ enumerator), 127
  - BLE\_SCAN\_FILTER\_ALLOW\_UND\_RPA\_DIR (C++ enumerator), 127
  - BLE\_SCAN\_FILTER\_ALLOW\_WLIST\_PRA\_DIR (C++ enumerator), 127
  - BLE\_SCAN\_TYPE\_ACTIVE (C++ enumerator), 127
  - BLE\_SCAN\_TYPE\_PASSIVE (C++ enumerator), 127
- ## C
- CONFIG\_HEAP\_TRACING\_STACK\_DEPTH (C macro), 566
- ## D
- DAC\_CHANNEL\_1 (C++ enumerator), 219
  - DAC\_CHANNEL\_1\_GPIO\_NUM (C macro), 219
  - DAC\_CHANNEL\_2 (C++ enumerator), 219
  - DAC\_CHANNEL\_2\_GPIO\_NUM (C macro), 219
  - DAC\_CHANNEL\_MAX (C++ enumerator), 219
  - dac\_channel\_t (C++ type), 219
  - DAC\_GPIO25\_CHANNEL (C macro), 219
  - DAC\_GPIO26\_CHANNEL (C macro), 219
  - dac\_i2s\_disable (C++ function), 218
  - dac\_i2s\_enable (C++ function), 218
  - dac\_output\_disable (C++ function), 218
  - dac\_output\_enable (C++ function), 218
  - dac\_output\_voltage (C++ function), 218
  - dac\_pad\_get\_io\_num (C++ function), 217

dmaworkaround\_cb\_t (C++ type), 336

## E

eAbortSleep (C++ enumerator), 474

eBlocked (C++ enumerator), 474

eDeleted (C++ enumerator), 474

eIncrement (C++ enumerator), 474

eNoAction (C++ enumerator), 474

eNoTasksWaitingTimeout (C++ enumerator), 475

eNotifyAction (C++ type), 474

eReady (C++ enumerator), 474

eRunning (C++ enumerator), 474

eSetBits (C++ enumerator), 474

eSetValueWithoutOverwrite (C++ enumerator), 474

eSetValueWithOverwrite (C++ enumerator), 474

eSleepModeStatus (C++ type), 474

ESP\_A2D\_AUDIO\_CFG\_EVT (C++ enumerator), 185

ESP\_A2D\_AUDIO\_STATE\_EVT (C++ enumerator), 185

ESP\_A2D\_AUDIO\_STATE\_REMOTE\_SUSPEND (C++ enumerator), 185

ESP\_A2D\_AUDIO\_STATE\_STARTED (C++ enumerator), 185

ESP\_A2D\_AUDIO\_STATE\_STOPPED (C++ enumerator), 185

esp\_a2d\_audio\_state\_t (C++ type), 185

esp\_a2d\_cb\_event\_t (C++ type), 185

esp\_a2d\_cb\_param\_t (C++ type), 183

esp\_a2d\_cb\_param\_t::a2d\_audio\_cfg\_param (C++ class), 183

esp\_a2d\_cb\_param\_t::a2d\_audio\_cfg\_param::mcc (C++ member), 183

esp\_a2d\_cb\_param\_t::a2d\_audio\_cfg\_param::remote\_register\_callback (C++ member), 183

esp\_a2d\_cb\_param\_t::a2d\_audio\_stat\_param (C++ class), 183

esp\_a2d\_cb\_param\_t::a2d\_audio\_stat\_param::remote\_bda (C++ member), 183

esp\_a2d\_cb\_param\_t::a2d\_audio\_stat\_param::state (C++ member), 183

esp\_a2d\_cb\_param\_t::a2d\_conn\_stat\_param (C++ class), 183

esp\_a2d\_cb\_param\_t::a2d\_conn\_stat\_param::adc\_rsn (C++ member), 183

esp\_a2d\_cb\_param\_t::a2d\_conn\_stat\_param::remote\_bda (C++ member), 183

esp\_a2d\_cb\_param\_t::a2d\_conn\_stat\_param::state (C++ member), 183

esp\_a2d\_cb\_param\_t::audio\_cfg (C++ member), 183

esp\_a2d\_cb\_param\_t::audio\_stat (C++ member), 183

esp\_a2d\_cb\_param\_t::conn\_stat (C++ member), 183

esp\_a2d\_cb\_t (C++ type), 184

ESP\_A2D\_CIE\_LEN\_ATRAC (C macro), 184

ESP\_A2D\_CIE\_LEN\_M12 (C macro), 184

ESP\_A2D\_CIE\_LEN\_M24 (C macro), 184

ESP\_A2D\_CIE\_LEN\_SBC (C macro), 184

ESP\_A2D\_CONNECTION\_STATE\_CONNECTED (C++ enumerator), 185

ESP\_A2D\_CONNECTION\_STATE\_CONNECTING (C++ enumerator), 185

ESP\_A2D\_CONNECTION\_STATE\_DISCONNECTED (C++ enumerator), 185

ESP\_A2D\_CONNECTION\_STATE\_DISCONNECTING (C++ enumerator), 185

ESP\_A2D\_CONNECTION\_STATE\_EVT (C++ enumerator), 185

esp\_a2d\_connection\_state\_t (C++ type), 185

esp\_a2d\_data\_cb\_t (C++ type), 184

ESP\_A2D\_DISC\_RSN\_ABNORMAL (C++ enumerator), 185

ESP\_A2D\_DISC\_RSN\_NORMAL (C++ enumerator), 185

esp\_a2d\_disc\_rsn\_t (C++ type), 185

esp\_a2d\_mcc\_t (C++ class), 184

esp\_a2d\_mcc\_t::cie (C++ member), 184

esp\_a2d\_mcc\_t::type (C++ member), 184

ESP\_A2D\_MCT\_ATRAC (C macro), 184

ESP\_A2D\_MCT\_M12 (C macro), 184

ESP\_A2D\_MCT\_M24 (C macro), 184

ESP\_A2D\_MCT\_NON\_A2DP (C macro), 184

ESP\_A2D\_MCT\_SBC (C macro), 184

esp\_a2d\_mct\_t (C++ type), 184

esp\_a2d\_register\_callback (C++ function), 181

esp\_a2d\_register\_data\_callback (C++ function), 181

esp\_a2d\_sink\_connect (C++ function), 182

esp\_a2d\_sink\_deinit (C++ function), 182

esp\_a2d\_sink\_disconnect (C++ function), 182

esp\_a2d\_sink\_init (C++ function), 182

esp\_adc\_cal\_characteristics\_t (C++ class), 215

esp\_adc\_cal\_characteristics\_t::adc\_num (C++ member), 215

esp\_adc\_cal\_characteristics\_t::atten (C++ member), 215

esp\_adc\_cal\_characteristics\_t::bit\_width (C++ member), 215

esp\_adc\_cal\_characteristics\_t::coeff\_a (C++ member), 215

esp\_adc\_cal\_characteristics\_t::coeff\_b (C++ member), 215

esp\_adc\_cal\_characteristics\_t::high\_curve

- (C++ member), 215
- esp\_adc\_cal\_characteristics\_t::low\_curve (C++ member), 215
- esp\_adc\_cal\_characteristics\_t::vref (C++ member), 215
- esp\_adc\_cal\_characterize (C++ function), 214
- esp\_adc\_cal\_check\_efuse (C++ function), 213
- esp\_adc\_cal\_get\_voltage (C++ function), 214
- esp\_adc\_cal\_raw\_to\_voltage (C++ function), 214
- ESP\_ADC\_CAL\_VAL\_DEFAULT\_VREF (C++ enumerator), 215
- ESP\_ADC\_CAL\_VAL\_EFUSE\_TP (C++ enumerator), 215
- ESP\_ADC\_CAL\_VAL\_EFUSE\_VREF (C++ enumerator), 215
- esp\_adc\_cal\_value\_t (C++ type), 215
- ESP\_APP\_ID\_MAX (C macro), 100
- ESP\_APP\_ID\_MIN (C macro), 100
- esp\_apprtrace\_buffer\_get (C++ function), 588
- esp\_apprtrace\_buffer\_put (C++ function), 588
- esp\_apprtrace\_dest\_t (C++ type), 592
- ESP\_APPTRACE\_DEST\_TRAX (C++ enumerator), 592
- ESP\_APPTRACE\_DEST\_UART0 (C++ enumerator), 592
- esp\_apprtrace\_down\_buffer\_config (C++ function), 588
- esp\_apprtrace\_down\_buffer\_get (C++ function), 590
- esp\_apprtrace\_down\_buffer\_put (C++ function), 590
- esp\_apprtrace\_fclose (C++ function), 591
- esp\_apprtrace\_flush (C++ function), 589
- esp\_apprtrace\_flush\_nolock (C++ function), 590
- esp\_apprtrace\_fopen (C++ function), 591
- esp\_apprtrace\_fread (C++ function), 591
- esp\_apprtrace\_fseek (C++ function), 592
- esp\_apprtrace\_fstop (C++ function), 592
- esp\_apprtrace\_ftell (C++ function), 592
- esp\_apprtrace\_fwrite (C++ function), 591
- esp\_apprtrace\_host\_is\_connected (C++ function), 591
- esp\_apprtrace\_init (C++ function), 588
- esp\_apprtrace\_read (C++ function), 590
- esp\_apprtrace\_vprintf (C++ function), 589
- esp\_apprtrace\_vprintf\_to (C++ function), 589
- esp\_apprtrace\_write (C++ function), 589
- esp\_attr\_control\_t (C++ class), 130
- esp\_attr\_control\_t::auto\_rsp (C++ member), 130
- esp\_attr\_desc\_t (C++ class), 129
- esp\_attr\_desc\_t::length (C++ member), 130
- esp\_attr\_desc\_t::max\_length (C++ member), 130
- esp\_attr\_desc\_t::perm (C++ member), 130
- esp\_attr\_desc\_t::uuid\_length (C++ member), 130
- esp\_attr\_desc\_t::uuid\_p (C++ member), 130
- esp\_attr\_desc\_t::value (C++ member), 130
- esp\_attr\_value\_t (C++ class), 130
- esp\_attr\_value\_t::attr\_len (C++ member), 130
- esp\_attr\_value\_t::attr\_max\_len (C++ member), 130
- esp\_attr\_value\_t::attr\_value (C++ member), 130
- esp\_avrc\_ct\_cb\_event\_t (C++ type), 191
- esp\_avrc\_ct\_cb\_param\_t (C++ type), 188
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_change\_notify\_param (C++ class), 188
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_change\_notify\_param (C++ member), 188
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_change\_notify\_param (C++ member), 188
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_conn\_stat\_param (C++ class), 188
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_conn\_stat\_param::c (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_conn\_stat\_param::e (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_meta\_rsp\_param (C++ class), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_meta\_rsp\_param::att (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_meta\_rsp\_param::att (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_meta\_rsp\_param::att (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_psth\_rsp\_param (C++ class), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_psth\_rsp\_param::key (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_psth\_rsp\_param::key (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_psth\_rsp\_param::tl (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_rmt\_feats\_param (C++ class), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_rmt\_feats\_param::fe (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::avrc\_ct\_rmt\_feats\_param::re (C++ member), 189
- esp\_avrc\_ct\_cb\_param\_t::change\_ntf (C++ member), 188
- esp\_avrc\_ct\_cb\_param\_t::conn\_stat (C++ member), 188

esp\_avrc\_ct\_cb\_param\_t::meta\_rsp (C++ member), 188  
 esp\_avrc\_ct\_cb\_param\_t::psth\_rsp (C++ member), 188  
 esp\_avrc\_ct\_cb\_param\_t::rmt\_feats (C++ member), 188  
 esp\_avrc\_ct\_cb\_t (C++ type), 189  
 ESP\_AVRC\_CT\_CHANGE\_NOTIFY\_EVT (C++ enumerator), 191  
 ESP\_AVRC\_CT\_CONNECTION\_STATE\_EVT (C++ enumerator), 191  
 esp\_avrc\_ct\_deinit (C++ function), 186  
 esp\_avrc\_ct\_init (C++ function), 186  
 ESP\_AVRC\_CT\_METADATA\_RSP\_EVT (C++ enumerator), 191  
 ESP\_AVRC\_CT\_PASSTHROUGH\_RSP\_EVT (C++ enumerator), 191  
 ESP\_AVRC\_CT\_PLAY\_STATUS\_RSP\_EVT (C++ enumerator), 191  
 esp\_avrc\_ct\_register\_callback (C++ function), 186  
 ESP\_AVRC\_CT\_REMOTE\_FEATURES\_EVT (C++ enumerator), 191  
 esp\_avrc\_ct\_send\_metadata\_cmd (C++ function), 187  
 esp\_avrc\_ct\_send\_passthrough\_cmd (C++ function), 188  
 esp\_avrc\_ct\_send\_register\_notification\_cmd (C++ function), 187  
 esp\_avrc\_ct\_send\_set\_player\_value\_cmd (C++ function), 187  
 ESP\_AVRC\_FEAT\_ADV\_CTRL (C++ enumerator), 190  
 ESP\_AVRC\_FEAT\_BROWSE (C++ enumerator), 190  
 ESP\_AVRC\_FEAT\_META\_DATA (C++ enumerator), 190  
 ESP\_AVRC\_FEAT\_RCCT (C++ enumerator), 190  
 ESP\_AVRC\_FEAT\_RCTG (C++ enumerator), 190  
 ESP\_AVRC\_FEAT\_VENDOR (C++ enumerator), 190  
 esp\_avrc\_features\_t (C++ type), 190  
 ESP\_AVRC\_MD\_ATTR\_ALBUM (C++ enumerator), 191  
 ESP\_AVRC\_MD\_ATTR\_ARTIST (C++ enumerator), 191  
 ESP\_AVRC\_MD\_ATTR\_GENRE (C++ enumerator), 191  
 esp\_avrc\_md\_attr\_mask\_t (C++ type), 191  
 ESP\_AVRC\_MD\_ATTR\_NUM\_TRACKS (C++ enumerator), 191  
 ESP\_AVRC\_MD\_ATTR\_PLAYING\_TIME (C++ enumerator), 191  
 ESP\_AVRC\_MD\_ATTR\_TITLE (C++ enumerator), 191  
 ESP\_AVRC\_MD\_ATTR\_TRACK\_NUM (C++ enumerator), 191  
 esp\_avrc\_ps\_attr\_ids\_t (C++ type), 192  
 esp\_avrc\_ps\_eq\_value\_ids\_t (C++ type), 192  
 ESP\_AVRC\_PS\_EQUALIZER (C++ enumerator), 192  
 ESP\_AVRC\_PS\_EQUALIZER\_OFF (C++ enumerator), 192  
 ESP\_AVRC\_PS\_EQUALIZER\_ON (C++ enumerator), 192  
 ESP\_AVRC\_PS\_MAX\_ATTR (C++ enumerator), 192  
 ESP\_AVRC\_PS\_REPEAT\_GROUP (C++ enumerator), 192  
 ESP\_AVRC\_PS\_REPEAT\_MODE (C++ enumerator), 192  
 ESP\_AVRC\_PS\_REPEAT\_OFF (C++ enumerator), 192  
 ESP\_AVRC\_PS\_REPEAT\_SINGLE (C++ enumerator), 192  
 esp\_avrc\_ps\_rpt\_value\_ids\_t (C++ type), 192  
 ESP\_AVRC\_PS\_SCAN\_ALL (C++ enumerator), 193  
 ESP\_AVRC\_PS\_SCAN\_GROUP (C++ enumerator), 193  
 ESP\_AVRC\_PS\_SCAN\_MODE (C++ enumerator), 192  
 ESP\_AVRC\_PS\_SCAN\_OFF (C++ enumerator), 193  
 esp\_avrc\_ps\_scn\_value\_ids\_t (C++ type), 193  
 esp\_avrc\_ps\_shf\_value\_ids\_t (C++ type), 192  
 ESP\_AVRC\_PS\_SHUFFLE\_ALL (C++ enumerator), 193  
 ESP\_AVRC\_PS\_SHUFFLE\_GROUP (C++ enumerator), 193  
 ESP\_AVRC\_PS\_SHUFFLE\_MODE (C++ enumerator), 192  
 ESP\_AVRC\_PS\_SHUFFLE\_OFF (C++ enumerator), 192  
 ESP\_AVRC\_PT\_CMD\_BACKWARD (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_FAST\_FORWARD (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_FORWARD (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_PAUSE (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_PLAY (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_REWIND (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_STATE\_PRESSED (C++ enumerator), 190  
 ESP\_AVRC\_PT\_CMD\_STATE\_RELEASED (C++ enumerator), 190  
 esp\_avrc\_pt\_cmd\_state\_t (C++ type), 190  
 ESP\_AVRC\_PT\_CMD\_STOP (C++ enumerator), 190  
 esp\_avrc\_pt\_cmd\_t (C++ type), 190  
 ESP\_AVRC\_RN\_APP\_SETTING\_CHANGE (C++ enumerator), 192  
 ESP\_AVRC\_RN\_BATTERY\_STATUS\_CHANGE (C++ enumerator), 192  
 esp\_avrc\_rn\_event\_ids\_t (C++ type), 191  
 ESP\_AVRC\_RN\_MAX\_EVT (C++ enumerator), 192  
 ESP\_AVRC\_RN\_PLAY\_POS\_CHANGED (C++ enumerator), 192  
 ESP\_AVRC\_RN\_PLAY\_STATUS\_CHANGE (C++ enumerator), 191  
 ESP\_AVRC\_RN\_SYSTEM\_STATUS\_CHANGE (C++

- enumerator*), 192
- ESP\_AVRC\_RN\_TRACK\_CHANGE (C++ *enumerator*), 191
- ESP\_AVRC\_RN\_TRACK\_REACHED\_END (C++ *enumerator*), 191
- ESP\_AVRC\_RN\_TRACK\_REACHED\_START (C++ *enumerator*), 191
- esp\_base\_mac\_addr\_set (C++ *function*), 606
- ESP\_BD\_ADDR\_HEX (C *macro*), 100
- ESP\_BD\_ADDR\_LEN (C *macro*), 100
- ESP\_BD\_ADDR\_STR (C *macro*), 100
- esp\_bd\_addr\_t (C++ *type*), 101
- ESP\_BLE\_AD\_MANUFACTURER\_SPECIFIC\_TYPE (C++ *enumerator*), 126
- ESP\_BLE\_AD\_TYPE\_128SERVICE\_DATA (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_128SOL\_SRV\_UUID (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_128SRV\_CMPL (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_128SRV\_PART (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_16SRV\_CMPL (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_16SRV\_PART (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_32SERVICE\_DATA (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_32SOL\_SRV\_UUID (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_32SRV\_CMPL (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_32SRV\_PART (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_ADV\_INT (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_APPEARANCE (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_CHAN\_MAP\_UPDATE (C++ *enumerator*), 126
- ESP\_BLE\_AD\_TYPE\_DEV\_CLASS (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_FLAG (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_INDOOR\_POSITION (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_INT\_RANGE (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_LE\_DEV\_ADDR (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_LE\_ROLE (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_LE\_SECURE\_CONFIRM (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_LE\_SECURE\_RANDOM (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_LE\_SUPPORT\_FEATURE (C++ *enumerator*), 126
- ESP\_BLE\_AD\_TYPE\_NAME\_CMPL (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_NAME\_SHORT (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_PUBLIC\_TARGET (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_RANDOM\_TARGET (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_SERVICE\_DATA (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_SM\_OOB\_FLAG (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_SM\_TK (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_SOL\_SRV\_UUID (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_SPAIR\_C256 (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_SPAIR\_R256 (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_TRANS\_DISC\_DATA (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_TX\_PWR (C++ *enumerator*), 125
- ESP\_BLE\_AD\_TYPE\_URI (C++ *enumerator*), 125
- esp\_ble\_addr\_type\_t (C++ *type*), 102
- esp\_ble\_adv\_channel\_t (C++ *type*), 126
- ESP\_BLE\_ADV\_DATA\_LEN\_MAX (C *macro*), 123
- esp\_ble\_adv\_data\_t (C++ *class*), 117
- esp\_ble\_adv\_data\_t::appearance (C++ *member*), 117
- esp\_ble\_adv\_data\_t::flag (C++ *member*), 118
- esp\_ble\_adv\_data\_t::include\_name (C++ *member*), 117
- esp\_ble\_adv\_data\_t::include\_txpower (C++ *member*), 117
- esp\_ble\_adv\_data\_t::manufacturer\_len (C++ *member*), 117
- esp\_ble\_adv\_data\_t::max\_interval (C++ *member*), 117
- esp\_ble\_adv\_data\_t::min\_interval (C++ *member*), 117
- esp\_ble\_adv\_data\_t::p\_manufacturer\_data (C++ *member*), 117
- esp\_ble\_adv\_data\_t::p\_service\_data (C++ *member*), 117
- esp\_ble\_adv\_data\_t::p\_service\_uuid (C++ *member*), 117
- esp\_ble\_adv\_data\_t::service\_data\_len (C++ *member*), 117
- esp\_ble\_adv\_data\_t::service\_uuid\_len (C++ *member*), 117
- esp\_ble\_adv\_data\_t::set\_scan\_rsp (C++

- member*), 117
- esp\_ble\_adv\_data\_type (C++ type), 125
- esp\_ble\_adv\_filter\_t (C++ type), 126
- ESP\_BLE\_ADV\_FLAG\_BREDR\_NOT\_SPT (C macro), 122
- ESP\_BLE\_ADV\_FLAG\_DMT\_CONTROLLER\_SPT (C macro), 122
- ESP\_BLE\_ADV\_FLAG\_DMT\_HOST\_SPT (C macro), 122
- ESP\_BLE\_ADV\_FLAG\_GEN\_DISC (C macro), 122
- ESP\_BLE\_ADV\_FLAG\_LIMIT\_DISC (C macro), 122
- ESP\_BLE\_ADV\_FLAG\_NON\_LIMIT\_DISC (C macro), 122
- esp\_ble\_adv\_params\_t (C++ class), 116
- esp\_ble\_adv\_params\_t::adv\_filter\_policy (C++ member), 117
- esp\_ble\_adv\_params\_t::adv\_int\_max (C++ member), 116
- esp\_ble\_adv\_params\_t::adv\_int\_min (C++ member), 116
- esp\_ble\_adv\_params\_t::adv\_type (C++ member), 117
- esp\_ble\_adv\_params\_t::channel\_map (C++ member), 117
- esp\_ble\_adv\_params\_t::own\_addr\_type (C++ member), 117
- esp\_ble\_adv\_params\_t::peer\_addr (C++ member), 117
- esp\_ble\_adv\_params\_t::peer\_addr\_type (C++ member), 117
- esp\_ble\_adv\_type\_t (C++ type), 126
- esp\_ble\_auth\_cmpl\_t (C++ class), 121
- esp\_ble\_auth\_cmpl\_t::addr\_type (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::bd\_addr (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::dev\_type (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::fail\_reason (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::key (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::key\_present (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::key\_type (C++ member), 122
- esp\_ble\_auth\_cmpl\_t::success (C++ member), 122
- esp\_ble\_auth\_req\_t (C++ type), 123
- esp\_ble\_bond\_dev\_t (C++ class), 121
- esp\_ble\_bond\_dev\_t::bd\_addr (C++ member), 121
- esp\_ble\_bond\_dev\_t::bond\_key (C++ member), 121
- esp\_ble\_bond\_key\_info\_t (C++ class), 120
- esp\_ble\_bond\_key\_info\_t::key\_mask (C++ member), 121
- esp\_ble\_bond\_key\_info\_t::pcsrk\_key (C++ member), 121
- esp\_ble\_bond\_key\_info\_t::penc\_key (C++ member), 121
- esp\_ble\_bond\_key\_info\_t::pid\_key (C++ member), 121
- esp\_ble\_confirm\_reply (C++ function), 109
- ESP\_BLE\_CONN\_INT\_MAX (C macro), 100
- ESP\_BLE\_CONN\_INT\_MIN (C macro), 100
- ESP\_BLE\_CONN\_LATENCY\_MAX (C macro), 100
- ESP\_BLE\_CONN\_PARAM\_UNDEF (C macro), 100
- ESP\_BLE\_CONN\_SUP\_TOUT\_MAX (C macro), 100
- ESP\_BLE\_CONN\_SUP\_TOUT\_MIN (C macro), 100
- esp\_ble\_conn\_update\_params\_t (C++ class), 118
- esp\_ble\_conn\_update\_params\_t::bda (C++ member), 118
- esp\_ble\_conn\_update\_params\_t::latency (C++ member), 118
- esp\_ble\_conn\_update\_params\_t::max\_int (C++ member), 118
- esp\_ble\_conn\_update\_params\_t::min\_int (C++ member), 118
- esp\_ble\_conn\_update\_params\_t::timeout (C++ member), 118
- ESP\_BLE\_CSR\_KEY\_MASK (C macro), 100
- ESP\_BLE\_ENC\_KEY\_MASK (C macro), 100
- ESP\_BLE\_EVT\_CONN\_ADV (C++ enumerator), 128
- ESP\_BLE\_EVT\_CONN\_DIR\_ADV (C++ enumerator), 128
- ESP\_BLE\_EVT\_DISC\_ADV (C++ enumerator), 128
- ESP\_BLE\_EVT\_NON\_CONN\_ADV (C++ enumerator), 128
- ESP\_BLE\_EVT\_SCAN\_RSP (C++ enumerator), 128
- esp\_ble\_evt\_type\_t (C++ type), 128
- esp\_ble\_gap\_cb\_param\_t (C++ type), 111
- esp\_ble\_gap\_cb\_param\_t::adv\_data\_cmpl (C++ member), 111
- esp\_ble\_gap\_cb\_param\_t::adv\_data\_raw\_cmpl (C++ member), 111
- esp\_ble\_gap\_cb\_param\_t::adv\_start\_cmpl (C++ member), 111
- esp\_ble\_gap\_cb\_param\_t::adv\_stop\_cmpl (C++ member), 112
- esp\_ble\_gap\_cb\_param\_t::ble\_adv\_data\_cmpl\_evt\_param (C++ class), 112
- esp\_ble\_gap\_cb\_param\_t::ble\_adv\_data\_cmpl\_evt\_param (C++ member), 112
- esp\_ble\_gap\_cb\_param\_t::ble\_adv\_data\_raw\_cmpl\_evt\_param (C++ class), 112
- esp\_ble\_gap\_cb\_param\_t::ble\_adv\_data\_raw\_cmpl\_evt\_param (C++ member), 112



esp\_ble\_gap\_cb\_param\_t::ble\_update\_conn\_params (C++ member), 116  
 esp\_ble\_gap\_cb\_param\_t::ble\_update\_whitelist\_cmpl (C++ class), 116  
 esp\_ble\_gap\_cb\_param\_t::ble\_update\_whitelist\_cmpl (C++ member), 116  
 esp\_ble\_gap\_cb\_param\_t::ble\_update\_whitelist\_cmpl (C++ member), 116  
 esp\_ble\_gap\_cb\_param\_t::clear\_bond\_dev\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::get\_bond\_dev\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::local\_privacy\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::pkt\_data\_lenth\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::read\_rssi\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::remove\_bond\_dev\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::scan\_param\_cmpl (C++ member), 111  
 esp\_ble\_gap\_cb\_param\_t::scan\_rsp\_data\_cmpl (C++ member), 111  
 esp\_ble\_gap\_cb\_param\_t::scan\_rsp\_data\_raw\_cmpl (C++ member), 111  
 esp\_ble\_gap\_cb\_param\_t::scan\_rst (C++ member), 111  
 esp\_ble\_gap\_cb\_param\_t::scan\_start\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::scan\_stop\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::set\_rand\_addr\_cmpl (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::update\_conn\_params (C++ member), 112  
 esp\_ble\_gap\_cb\_param\_t::update\_whitelists\_cmpl (C++ member), 112  
 esp\_ble\_gap\_config\_adv\_data (C++ function), 105  
 esp\_ble\_gap\_config\_adv\_data\_raw (C++ function), 108  
 esp\_ble\_gap\_config\_local\_privacy (C++ function), 106  
 esp\_ble\_gap\_config\_scan\_rsp\_data\_raw (C++ function), 108  
 esp\_ble\_gap\_disconnect (C++ function), 110  
 esp\_ble\_gap\_get\_whitelist\_size (C++ function), 107  
 esp\_ble\_gap\_read\_rssi (C++ function), 108  
 esp\_ble\_gap\_register\_callback (C++ function), 104  
 esp\_ble\_gap\_security\_rsp (C++ function), 109  
 esp\_ble\_gap\_set\_device\_name (C++ function), 107  
 esp\_ble\_gap\_set\_pkt\_data\_len (C++ function), 106  
 esp\_ble\_gap\_set\_prefer\_conn\_params (C++ function), 107  
 esp\_ble\_gap\_set\_rand\_addr (C++ function), 106  
 esp\_ble\_gap\_set\_scan\_params (C++ function), 105  
 esp\_ble\_gap\_set\_security\_param (C++ function), 108  
 esp\_ble\_gap\_start\_advertising (C++ function), 105  
 esp\_ble\_gap\_start\_scanning (C++ function), 105  
 esp\_ble\_gap\_stop\_advertising (C++ function), 106  
 esp\_ble\_gap\_stop\_scanning (C++ function), 106  
 esp\_ble\_gap\_update\_conn\_params (C++ function), 106  
 esp\_ble\_gap\_update\_whitelist (C++ function), 106  
 esp\_ble\_gattc\_app\_register (C++ function), 153  
 esp\_ble\_gattc\_app\_unregister (C++ function), 153  
 esp\_ble\_gattc\_cache\_refresh (C++ function), 162  
 esp\_ble\_gattc\_cb\_param\_t (C++ type), 163  
 esp\_ble\_gattc\_cb\_param\_t::cfg\_mtu (C++ member), 163  
 esp\_ble\_gattc\_cb\_param\_t::close (C++ member), 163  
 esp\_ble\_gattc\_cb\_param\_t::congest (C++ member), 163  
 esp\_ble\_gattc\_cb\_param\_t::connect (C++ member), 163  
 esp\_ble\_gattc\_cb\_param\_t::disconnect (C++ member), 163  
 esp\_ble\_gattc\_cb\_param\_t::exec\_cmpl (C++ member), 163  
 esp\_ble\_gattc\_cb\_param\_t::gattc\_cfg\_mtu\_evt\_param (C++ class), 163  
 esp\_ble\_gattc\_cb\_param\_t::gattc\_cfg\_mtu\_evt\_param (C++ member), 164  
 esp\_ble\_gattc\_cb\_param\_t::gattc\_cfg\_mtu\_evt\_param (C++ member), 164  
 esp\_ble\_gattc\_cb\_param\_t::gattc\_cfg\_mtu\_evt\_param (C++ member), 164  
 esp\_ble\_gattc\_cb\_param\_t::gattc\_close\_evt\_param (C++ class), 164  
 esp\_ble\_gattc\_cb\_param\_t::gattc\_close\_evt\_param (C++ member), 164









esp\_ble\_gatts\_open (C++ function), 144  
 esp\_ble\_gatts\_register\_callback (C++ function), 140  
 esp\_ble\_gatts\_send\_indicate (C++ function), 142  
 esp\_ble\_gatts\_send\_response (C++ function), 143  
 esp\_ble\_gatts\_set\_attr\_value (C++ function), 143  
 esp\_ble\_gatts\_start\_service (C++ function), 142  
 esp\_ble\_gatts\_stop\_service (C++ function), 142  
 esp\_ble\_get\_bond\_device\_list (C++ function), 110  
 esp\_ble\_get\_bond\_device\_num (C++ function), 110  
 ESP\_BLE\_ID\_KEY\_MASK (C macro), 100  
 esp\_ble\_io\_cap\_t (C++ type), 123  
 ESP\_BLE\_IS\_VALID\_PARAM (C macro), 100  
 esp\_ble\_key\_mask\_t (C++ type), 101  
 esp\_ble\_key\_t (C++ class), 121  
 esp\_ble\_key\_t::bd\_addr (C++ member), 121  
 esp\_ble\_key\_t::key\_type (C++ member), 121  
 esp\_ble\_key\_t::p\_key\_value (C++ member), 121  
 esp\_ble\_key\_type\_t (C++ type), 123  
 esp\_ble\_key\_value\_t (C++ type), 110  
 esp\_ble\_key\_value\_t::lcsr\_key (C++ member), 111  
 esp\_ble\_key\_value\_t::lenc\_key (C++ member), 111  
 esp\_ble\_key\_value\_t::pcsrk\_key (C++ member), 111  
 esp\_ble\_key\_value\_t::penc\_key (C++ member), 111  
 esp\_ble\_key\_value\_t::pid\_key (C++ member), 111  
 esp\_ble\_lcsr\_keys (C++ class), 120  
 esp\_ble\_lcsr\_keys::counter (C++ member), 120  
 esp\_ble\_lcsr\_keys::csrk (C++ member), 120  
 esp\_ble\_lcsr\_keys::div (C++ member), 120  
 esp\_ble\_lcsr\_keys::sec\_level (C++ member), 120  
 esp\_ble\_lenc\_keys\_t (C++ class), 119  
 esp\_ble\_lenc\_keys\_t::div (C++ member), 120  
 esp\_ble\_lenc\_keys\_t::key\_size (C++ member), 120  
 esp\_ble\_lenc\_keys\_t::ltk (C++ member), 120  
 esp\_ble\_lenc\_keys\_t::sec\_level (C++ member), 120  
 ESP\_BLE\_LINK\_KEY\_MASK (C macro), 100  
 esp\_ble\_local\_id\_keys\_t (C++ class), 121  
 esp\_ble\_local\_id\_keys\_t::dhk (C++ member), 121  
 esp\_ble\_local\_id\_keys\_t::ir (C++ member), 121  
 esp\_ble\_local\_id\_keys\_t::irk (C++ member), 121  
 esp\_ble\_passkey\_reply (C++ function), 109  
 esp\_ble\_pcsr\_keys\_t (C++ class), 119  
 esp\_ble\_pcsr\_keys\_t::counter (C++ member), 119  
 esp\_ble\_pcsr\_keys\_t::csrk (C++ member), 119  
 esp\_ble\_pcsr\_keys\_t::sec\_level (C++ member), 119  
 esp\_ble\_penc\_keys\_t (C++ class), 119  
 esp\_ble\_penc\_keys\_t::ediv (C++ member), 119  
 esp\_ble\_penc\_keys\_t::key\_size (C++ member), 119  
 esp\_ble\_penc\_keys\_t::ltk (C++ member), 119  
 esp\_ble\_penc\_keys\_t::rand (C++ member), 119  
 esp\_ble\_penc\_keys\_t::sec\_level (C++ member), 119  
 esp\_ble\_pid\_keys\_t (C++ class), 119  
 esp\_ble\_pid\_keys\_t::addr\_type (C++ member), 119  
 esp\_ble\_pid\_keys\_t::irk (C++ member), 119  
 esp\_ble\_pid\_keys\_t::static\_addr (C++ member), 119  
 esp\_ble\_pkt\_data\_length\_params\_t (C++ class), 118  
 esp\_ble\_pkt\_data\_length\_params\_t::rx\_len (C++ member), 119  
 esp\_ble\_pkt\_data\_length\_params\_t::tx\_len (C++ member), 119  
 esp\_ble\_remove\_bond\_device (C++ function), 109  
 esp\_ble\_resolve\_adv\_data (C++ function), 108  
 esp\_ble\_scan\_duplicate\_t (C++ type), 127  
 esp\_ble\_scan\_filter\_t (C++ type), 127  
 ESP\_BLE\_SCAN\_PARAM\_UNDEF (C macro), 100  
 esp\_ble\_scan\_params\_t (C++ class), 118  
 esp\_ble\_scan\_params\_t::own\_addr\_type (C++ member), 118  
 esp\_ble\_scan\_params\_t::scan\_duplicate (C++ member), 118  
 esp\_ble\_scan\_params\_t::scan\_filter\_policy (C++ member), 118  
 esp\_ble\_scan\_params\_t::scan\_interval (C++ member), 118  
 esp\_ble\_scan\_params\_t::scan\_type (C++ member), 118  
 esp\_ble\_scan\_params\_t::scan\_window (C++

member), 118  
ESP\_BLE\_SCAN\_RSP\_DATA\_LEN\_MAX (C macro), 123  
esp\_ble\_scan\_type\_t (C++ type), 127  
esp\_ble\_sec\_act\_t (C++ type), 126  
ESP\_BLE\_SEC\_ENCRYPT (C++ enumerator), 126  
ESP\_BLE\_SEC\_ENCRYPT\_MITM (C++ enumerator), 126  
ESP\_BLE\_SEC\_ENCRYPT\_NO\_MITM (C++ enumerator), 126  
esp\_ble\_sec\_key\_notif\_t (C++ class), 120  
esp\_ble\_sec\_key\_notif\_t::bd\_addr (C++ member), 120  
esp\_ble\_sec\_key\_notif\_t::passkey (C++ member), 120  
esp\_ble\_sec\_req\_t (C++ class), 120  
esp\_ble\_sec\_req\_t::bd\_addr (C++ member), 120  
esp\_ble\_sec\_t (C++ type), 111  
esp\_ble\_sec\_t::auth\_cmpl (C++ member), 111  
esp\_ble\_sec\_t::ble\_id\_keys (C++ member), 111  
esp\_ble\_sec\_t::ble\_key (C++ member), 111  
esp\_ble\_sec\_t::ble\_req (C++ member), 111  
esp\_ble\_sec\_t::key\_notif (C++ member), 111  
esp\_ble\_set\_encryption (C++ function), 109  
ESP\_BLE\_SM\_AUTHEN\_REQ\_MODE (C++ enumerator), 126  
ESP\_BLE\_SM\_IOCTL\_MODE (C++ enumerator), 126  
ESP\_BLE\_SM\_MAX\_KEY\_SIZE (C++ enumerator), 127  
esp\_ble\_sm\_param\_t (C++ type), 126  
ESP\_BLE\_SM\_PASSKEY (C++ enumerator), 126  
ESP\_BLE\_SM\_SET\_INIT\_KEY (C++ enumerator), 126  
ESP\_BLE\_SM\_SET\_RSP\_KEY (C++ enumerator), 127  
ESP\_BLE\_WHITELIST\_ADD (C++ enumerator), 128  
ESP\_BLE\_WHITELIST\_REMOVE (C++ enumerator), 128  
esp\_ble\_wl\_operation\_t (C++ type), 128  
esp\_bluedroid\_deinit (C++ function), 103  
esp\_bluedroid\_disable (C++ function), 102  
esp\_bluedroid\_enable (C++ function), 102  
esp\_bluedroid\_get\_status (C++ function), 102  
esp\_bluedroid\_init (C++ function), 103  
ESP\_BLUEDROID\_STATUS\_CHECK (C macro), 100  
ESP\_BLUEDROID\_STATUS\_ENABLED (C++ enumerator), 103  
ESP\_BLUEDROID\_STATUS\_INITIALIZED (C++ enumerator), 103  
esp\_bluedroid\_status\_t (C++ type), 103  
ESP\_BLUEDROID\_STATUS\_UNINITIALIZED (C++ enumerator), 103  
esp\_blufi\_callbacks\_t (C++ class), 177

esp\_blufi\_callbacks\_t::checksum\_func (C++ member), 177  
esp\_blufi\_callbacks\_t::decrypt\_func (C++ member), 177  
esp\_blufi\_callbacks\_t::encrypt\_func (C++ member), 177  
esp\_blufi\_callbacks\_t::event\_cb (C++ member), 177  
esp\_blufi\_callbacks\_t::negotiate\_data\_handler (C++ member), 177  
esp\_blufi\_cb\_event\_t (C++ type), 178  
esp\_blufi\_cb\_param\_t (C++ type), 172  
esp\_blufi\_cb\_param\_t::blufi\_connect\_evt\_param (C++ class), 173  
esp\_blufi\_cb\_param\_t::blufi\_connect\_evt\_param::conn (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_connect\_evt\_param::rem (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_connect\_evt\_param::serv (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_deinit\_finish\_evt\_param (C++ class), 173  
esp\_blufi\_cb\_param\_t::blufi\_deinit\_finish\_evt\_param (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_disconnect\_evt\_param (C++ class), 173  
esp\_blufi\_cb\_param\_t::blufi\_disconnect\_evt\_param (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_init\_finish\_evt\_param (C++ class), 173  
esp\_blufi\_cb\_param\_t::blufi\_init\_finish\_evt\_param (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_ca\_evt\_param (C++ class), 173  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_ca\_evt\_param::cert (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_ca\_evt\_param::cert (C++ member), 173  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_client\_cert\_evt\_param (C++ class), 173  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_client\_cert\_evt\_param (C++ member), 174  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_client\_cert\_evt\_param (C++ member), 174  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_client\_pkey\_evt\_param (C++ class), 174  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_client\_pkey\_evt\_param (C++ member), 174  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_client\_pkey\_evt\_param (C++ member), 174  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_server\_cert\_evt\_param (C++ class), 174  
esp\_blufi\_cb\_param\_t::blufi\_rcv\_server\_cert\_evt\_param (C++ member), 174

esp\_blufi\_cb\_param\_t::blufi\_recv\_server\_evt\_blufliparam\_t::blufi\_set\_wifi\_mode\_evt\_param  
 (C++ member), 174 (C++ class), 176  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_server\_key\_evt\_blufliparam\_t::blufi\_set\_wifi\_mode\_evt\_param  
 (C++ class), 174 (C++ member), 176  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_server\_key\_evt\_blufliparam\_t::ca (C++ member), 172  
 (C++ member), 174 esp\_blufi\_cb\_param\_t::client\_cert (C++  
 member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_server\_pkey\_evt\_blufliparam\_t::pkey\_len  
 (C++ member), 174 esp\_blufi\_cb\_param\_t::client\_pkey (C++  
 member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_softap\_auth\_mode\_evt\_blufliparam  
 (C++ class), 174 esp\_blufi\_cb\_param\_t::connect (C++ mem-  
 ber), 172 param::auth\_mode  
 (C++ member), 174 esp\_blufi\_cb\_param\_t::deinit\_finish  
 (C++ member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_softap\_channel\_evt\_blufliparam  
 (C++ class), 174 esp\_blufi\_cb\_param\_t::disconnect (C++  
 member), 172 param::channel  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::init\_finish (C++  
 member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_softap\_max\_conn\_evt\_blufliparam  
 (C++ class), 175 esp\_blufi\_cb\_param\_t::server\_cert (C++  
 member), 172 param::max\_conn\_num  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::server\_pkey (C++  
 member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_softap\_passwd\_evt\_blufliparam  
 (C++ class), 175 esp\_blufi\_cb\_param\_t::softap\_auth\_mode  
 (C++ member), 172 param::passwd  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::softap\_channel  
 (C++ member), 172 param::passwd\_len  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::softap\_max\_conn\_num  
 (C++ member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_softap\_ssids\_evt\_blufliparam  
 (C++ class), 175 esp\_blufi\_cb\_param\_t::softap\_passwd  
 (C++ member), 172 param::ssid  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::softap\_ssids (C++  
 member), 172 param::ssid\_len  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::sta\_bssid (C++  
 member), 172  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_sta\_bssid\_evt\_blufliparam  
 (C++ class), 175 esp\_blufi\_cb\_param\_t::sta\_passwd (C++  
 member), 172 param::bssid  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::sta\_ssids (C++ mem-  
 ber), 172 param::passwd  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::username (C++ mem-  
 ber), 172 param::passwd  
 (C++ member), 175 esp\_blufi\_cb\_param\_t::wifi\_mode (C++  
 member), 172 param::passwd\_len  
 (C++ member), 175 esp\_blufi\_checksum\_func\_t (C++ type), 178  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_sta\_ssids\_evt\_blufliparamlose (C++ function), 171  
 (C++ class), 175 esp\_blufi\_decrypt\_func\_t (C++ type), 178  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_sta\_ssids\_evt\_blufliparamFAILED (C++ enumerator),  
 (C++ member), 176 180  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_sta\_ssids\_evt\_blufliparamOK (C++ enumerator), 179  
 (C++ member), 176 esp\_blufi\_deinit\_state\_t (C++ type), 179  
 esp\_blufi\_cb\_param\_t::blufi\_recv\_username\_evt\_blufliparamncrypt\_func\_t (C++ type), 178  
 (C++ class), 176 ESP\_BLUFI\_EVENT\_BLE\_CONNECT (C++ enumera-  
 tor), 179 name  
 (C++ member), 176 ESP\_BLUFI\_EVENT\_BLE\_DISCONNECT (C++ enu-  
 mator), 179 name\_len  
 (C++ member), 176 esp\_blufi\_event\_cb\_t (C++ type), 177

ESP\_BLUFI\_EVENT\_DEAUTHENTICATE\_STA (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_DEINIT\_FINISH (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_GET\_WIFI\_STATUS (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_INIT\_FINISH (C++ enumerator), 178

ESP\_BLUFI\_EVENT\_RECV\_CA\_CERT (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_CLIENT\_CERT (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_CLIENT\_PRIV\_KEY (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SERVER\_CERT (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SERVER\_PRIV\_KEY (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SLAVE\_DISCONNECT\_BLE (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_AUTH\_MODE (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_CHANNEL (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_MAX\_CONN\_NUMES (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_PASSWD (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_SOFTAP\_SSID (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_STA\_BSSID (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_STA\_PASSWD (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_STA\_SSID (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_RECV\_USERNAME (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_REQ\_CONNECT\_TO\_AP (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_REQ\_DISCONNECT\_FROM\_AP (C++ enumerator), 179

ESP\_BLUFI\_EVENT\_SET\_WIFI\_OPMODE (C++ enumerator), 179

esp\_blufi\_extra\_info\_t (C++ class), 176

esp\_blufi\_extra\_info\_t::softap\_authmode (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_authmode\_esp (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_channel (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_channel\_set (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_max\_conn\_num (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_max\_conn\_num\_set (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_passwd (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_passwd\_len (C++ member), 177

esp\_blufi\_extra\_info\_t::softap\_ssid (C++ member), 176

esp\_blufi\_extra\_info\_t::softap\_ssid\_len (C++ member), 177

esp\_blufi\_extra\_info\_t::sta\_bssid (C++ member), 176

esp\_blufi\_extra\_info\_t::sta\_bssid\_set (C++ member), 176

esp\_blufi\_extra\_info\_t::sta\_passwd (C++ member), 176

esp\_blufi\_extra\_info\_t::sta\_passwd\_len (C++ member), 176

esp\_blufi\_extra\_info\_t::sta\_ssid (C++ member), 176

esp\_blufi\_extra\_info\_t::sta\_ssid\_len (C++ member), 176

esp\_blufi\_get\_version (C++ function), 171

ESP\_BLUFI\_INIT\_FAILED (C++ enumerator), 179

ESP\_BLUFI\_INIT\_OK (C++ enumerator), 179

esp\_blufi\_init\_state\_t (C++ type), 179

esp\_blufi\_negotiate\_data\_handler\_t (C++ type), 178

esp\_blufi\_profile\_deinit (C++ function), 171

esp\_blufi\_profile\_init (C++ function), 171

esp\_blufi\_register\_callbacks (C++ function), 171

esp\_blufi\_send\_wifi\_conn\_report (C++ function), 171

ESP\_BLUFI\_STA\_CONN\_FAIL (C++ enumerator), 179

esp\_blufi\_sta\_conn\_state\_t (C++ type), 179

ESP\_BLUFI\_STA\_CONN\_SUCCESS (C++ enumerator), 179

esp\_bt\_dev\_get\_address (C++ function), 104

esp\_bt\_dev\_set\_device\_name (C++ function), 104

esp\_bt\_dev\_type\_t (C++ type), 101

ESP\_BT\_DEVICE\_TYPE\_BLE (C++ enumerator), 102

ESP\_BT\_DEVICE\_TYPE\_BREDR (C++ enumerator), 101

ESP\_BT\_DEVICE\_TYPE\_DUMO (C++ enumerator), 102

esp\_bt\_gap\_set\_scan\_mode (C++ function), 180

ESP\_BT\_OCTET16\_LEN (C macro), 100

esp\_bt\_octet16\_t (C++ type), 101

ESP\_BT\_OCTET8\_LEN (C macro), 100

esp\_bt\_octet8\_t (C++ type), 101

ESP\_BT\_SCAN\_MODE\_CONNECTABLE (C++ enumerator), 180

ESP\_BT\_SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE (C++ enumerator), 181

ESP\_BT\_SCAN\_MODE\_NONE (C++ enumerator), 180

esp\_bt\_scan\_mode\_t (C++ type), 180

ESP\_BT\_STATUS\_AUTH\_FAILURE (C++ enumerator), 101

ESP\_BT\_STATUS\_AUTH\_REJECTED (C++ enumerator), 101

ESP\_BT\_STATUS\_BUSY (C++ enumerator), 101

ESP\_BT\_STATUS\_CONTROL\_LE\_DATA\_LEN\_UNSUPPORTED (C++ enumerator), 101

ESP\_BT\_STATUS\_DONE (C++ enumerator), 101

ESP\_BT\_STATUS\_ERR\_ILLEGAL\_PARAMETER\_FMT (C++ enumerator), 101

ESP\_BT\_STATUS\_FAIL (C++ enumerator), 101

ESP\_BT\_STATUS\_INVALID\_STATIC\_RAND\_ADDR (C++ enumerator), 101

ESP\_BT\_STATUS\_MEMORY\_FULL (C++ enumerator), 101

ESP\_BT\_STATUS\_NOMEM (C++ enumerator), 101

ESP\_BT\_STATUS\_NOT\_READY (C++ enumerator), 101

ESP\_BT\_STATUS\_PARAM\_OUT\_OF\_RANGE (C++ enumerator), 101

ESP\_BT\_STATUS\_PARM\_INVALID (C++ enumerator), 101

ESP\_BT\_STATUS\_PEER\_LE\_DATA\_LEN\_UNSUPPORTED (C++ enumerator), 101

ESP\_BT\_STATUS\_PENDING (C++ enumerator), 101

ESP\_BT\_STATUS\_RMT\_DEV\_DOWN (C++ enumerator), 101

ESP\_BT\_STATUS\_SUCCESS (C++ enumerator), 101

esp\_bt\_status\_t (C++ type), 101

ESP\_BT\_STATUS\_TIMEOUT (C++ enumerator), 101

ESP\_BT\_STATUS\_UNACCEPT\_CONN\_INTERVAL (C++ enumerator), 101

ESP\_BT\_STATUS\_UNHANDLED (C++ enumerator), 101

ESP\_BT\_STATUS\_UNSUPPORTED (C++ enumerator), 101

esp\_bt\_uuid\_t (C++ class), 99

esp\_bt\_uuid\_t::len (C++ member), 99

esp\_bt\_uuid\_t::uuid (C++ member), 99

esp\_deep\_sleep\_start (C++ function), 603

ESP\_DEFAULT\_GATT\_IF (C macro), 100

esp\_deregister\_freertos\_idle\_hook (C++ function), 546

esp\_deregister\_freertos\_idle\_hook\_for\_cp (C++ function), 546

esp\_deregister\_freertos\_tick\_hook (C++ function), 546

esp\_deregister\_freertos\_tick\_hook\_for\_cp (C++ function), 546

ESP\_EARLY\_LOGD (C macro), 586

ESP\_EARLY\_LOGE (C macro), 586

ESP\_EARLY\_LOGI (C macro), 586

ESP\_EARLY\_LOGV (C macro), 586

ESP\_EARLY\_LOGW (C macro), 586

esp\_efuse\_mac\_get\_custom (C++ function), 606

ESP\_ERR\_ESPNOW\_ARG (C macro), 97

ESP\_ERR\_ESPNOW\_BASE (C macro), 97

ESP\_ERR\_ESPNOW\_EXIST (C macro), 97

ESP\_ERR\_ESPNOW\_FULL (C macro), 97

ESP\_ERR\_ESPNOW\_IF (C macro), 97

ESP\_ERR\_ESPNOW\_INTERNAL (C macro), 97

ESP\_ERR\_ESPNOW\_NO\_MEM (C macro), 97

ESP\_ERR\_ESPNOW\_NOT\_FOUND (C macro), 97

ESP\_ERR\_ESPNOW\_NOT\_INIT (C macro), 97

ESP\_ERR\_FLASH\_BASE (C macro), 410

ESP\_ERR\_FLASH\_OP\_FAIL (C macro), 410

ESP\_ERR\_FLASH\_OP\_TIMEOUT (C macro), 410

ESP\_ERR\_NVS\_BASE (C macro), 430

ESP\_ERR\_NVS\_INVALID\_HANDLE (C macro), 430

ESP\_ERR\_NVS\_INVALID\_LENGTH (C macro), 430

ESP\_ERR\_NVS\_INVALID\_NAME (C macro), 430

ESP\_ERR\_NVS\_INVALID\_STATE (C macro), 430

ESP\_ERR\_NVS\_KEY\_TOO\_LONG (C macro), 430

ESP\_ERR\_NVS\_NO\_FREE\_PAGES (C macro), 430

ESP\_ERR\_NVS\_NOT\_ENOUGH\_SPACE (C macro), 430

ESP\_ERR\_NVS\_NOT\_FOUND (C macro), 430

ESP\_ERR\_NVS\_NOT\_INITIALIZED (C macro), 430

ESP\_ERR\_NVS\_PAGE\_FULL (C macro), 430

ESP\_ERR\_NVS\_PART\_NOT\_FOUND (C macro), 430

ESP\_ERR\_NVS\_READ\_ONLY (C macro), 430

ESP\_ERR\_NVS\_REMOVE\_FAILED (C macro), 430

ESP\_ERR\_NVS\_TYPE\_MISMATCH (C macro), 430

ESP\_ERR\_NVS\_VALUE\_TOO\_LONG (C macro), 430

ESP\_ERR\_OTA\_BASE (C macro), 610

ESP\_ERR\_OTA\_PARTITION\_CONFLICT (C macro), 610

ESP\_ERR\_OTA\_SELECT\_INFO\_INVALID (C macro), 610

ESP\_ERR\_OTA\_VALIDATE\_FAILED (C macro), 610

ESP\_ERR\_ULP\_BASE (C macro), 814

ESP\_ERR\_ULP\_BRANCH\_OUT\_OF\_RANGE (C macro), 815

ESP\_ERR\_ULP\_DUPLICATE\_LABEL (C macro), 815

ESP\_ERR\_ULP\_INVALID\_LOAD\_ADDR (C macro), 814

ESP\_ERR\_ULP\_SIZE\_TOO\_BIG (C macro), 814

ESP\_ERR\_ULP\_UNDEFINED\_LABEL (C macro), 815

ESP\_ERR\_WIFI\_ARG (C macro), 76

ESP\_ERR\_WIFI\_CONN (C macro), 76

ESP\_ERR\_WIFI\_FAIL (C macro), 76

ESP\_ERR\_WIFI\_IF (C macro), 76

ESP\_ERR\_WIFI\_MAC (C macro), 76

- ESP\_ERR\_WIFI\_MODE (*C macro*), 76
- ESP\_ERR\_WIFI\_NO\_MEM (*C macro*), 76
- ESP\_ERR\_WIFI\_NOT\_CONNECT (*C macro*), 77
- ESP\_ERR\_WIFI\_NOT\_INIT (*C macro*), 76
- ESP\_ERR\_WIFI\_NOT\_STARTED (*C macro*), 76
- ESP\_ERR\_WIFI\_NOT\_STOPPED (*C macro*), 76
- ESP\_ERR\_WIFI\_NOT\_SUPPORT (*C macro*), 76
- ESP\_ERR\_WIFI\_NV\_S (*C macro*), 76
- ESP\_ERR\_WIFI\_OK (*C macro*), 76
- ESP\_ERR\_WIFI\_PASSWORD (*C macro*), 76
- ESP\_ERR\_WIFI\_SSID (*C macro*), 76
- ESP\_ERR\_WIFI\_STATE (*C macro*), 76
- ESP\_ERR\_WIFI\_TIMEOUT (*C macro*), 76
- ESP\_ERR\_WIFI\_WAKE\_FAIL (*C macro*), 76
- ESP\_ERR\_WIFI\_WOULD\_BLOCK (*C macro*), 77
- esp\_esptouch\_set\_timeout (*C++ function*), 89
- esp\_eth\_disable (*C++ function*), 195
- esp\_eth\_enable (*C++ function*), 195
- esp\_eth\_free\_rx\_buf (*C++ function*), 196
- esp\_eth\_get\_mac (*C++ function*), 195
- esp\_eth\_init (*C++ function*), 194
- esp\_eth\_init\_internal (*C++ function*), 194
- esp\_eth\_set\_mac (*C++ function*), 196
- esp\_eth\_smi\_read (*C++ function*), 195
- esp\_eth\_smi\_wait\_set (*C++ function*), 196
- esp\_eth\_smi\_wait\_value (*C++ function*), 195
- esp\_eth\_smi\_write (*C++ function*), 195
- esp\_eth\_tx (*C++ function*), 194
- ESP\_EXT1\_WAKEUP\_ALL\_LOW (*C++ enumerator*), 601
- ESP\_EXT1\_WAKEUP\_ANY\_HIGH (*C++ enumerator*), 601
- esp\_flash\_encrypt\_check\_and\_update (*C++ function*), 417
- esp\_flash\_encrypt\_region (*C++ function*), 417
- esp\_flash\_encryption\_enabled (*C++ function*), 417
- esp\_flash\_write\_protect\_crypt\_cnt (*C++ function*), 417
- esp\_freertos\_idle\_cb\_t (*C++ type*), 546
- esp\_freertos\_tick\_cb\_t (*C++ type*), 546
- ESP\_GAP\_BLE\_ADD\_WHITELIST\_COMPLETE\_EVT (*C macro*), 123
- ESP\_GAP\_BLE\_ADV\_DATA\_RAW\_SET\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_ADV\_DATA\_SET\_COMPLETE\_EVT (*C++ enumerator*), 123
- ESP\_GAP\_BLE\_ADV\_START\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_ADV\_STOP\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_AUTH\_CMPL\_EVT (*C++ enumerator*), 124
- esp\_gap\_ble\_cb\_event\_t (*C++ type*), 123
- esp\_gap\_ble\_cb\_t (*C++ type*), 123
- ESP\_GAP\_BLE\_CLEAR\_BOND\_DEV\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_EVT\_MAX (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_GET\_BOND\_DEV\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_KEY\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_LOCAL\_ER\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_LOCAL\_IR\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_NC\_REQ\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_OOB\_REQ\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_PASSKEY\_NOTIF\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_PASSKEY\_REQ\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_READ\_RSSI\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_REMOVE\_BOND\_DEV\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SCAN\_PARAM\_SET\_COMPLETE\_EVT (*C++ enumerator*), 123
- ESP\_GAP\_BLE\_SCAN\_RESULT\_EVT (*C++ enumerator*), 123
- ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_RAW\_SET\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SCAN\_RSP\_DATA\_SET\_COMPLETE\_EVT (*C++ enumerator*), 123
- ESP\_GAP\_BLE\_SCAN\_START\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SCAN\_STOP\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SEC\_REQ\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SET\_LOCAL\_PRIVACY\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SET\_PKT\_LENGTH\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_SET\_STATIC\_RAND\_ADDR\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_UPDATE\_CONN\_PARAMS\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_BLE\_UPDATE\_WHITELIST\_COMPLETE\_EVT (*C++ enumerator*), 124
- ESP\_GAP\_SEARCH\_DI\_DISC\_CMPL\_EVT (*C++ enumerator*), 128
- ESP\_GAP\_SEARCH\_DISC\_BLE\_RES\_EVT (*C++ enumerator*), 128
- ESP\_GAP\_SEARCH\_DISC\_CMPL\_EVT (*C++ enumerator*), 128
- ESP\_GAP\_SEARCH\_DISC\_RES\_EVT (*C++ enumerator*), 128

- esp\_gap\_search\_evt\_t (C++ type), 127
- ESP\_GAP\_SEARCH\_INQ\_CMPL\_EVT (C++ enumerator), 128
- ESP\_GAP\_SEARCH\_INQ\_RES\_EVT (C++ enumerator), 128
- ESP\_GAP\_SEARCH\_SEARCH\_CANCEL\_CMPL\_EVT (C++ enumerator), 128
- ESP\_GATT\_ALREADY\_OPEN (C++ enumerator), 138
- ESP\_GATT\_APP\_RSP (C++ enumerator), 138
- ESP\_GATT\_ATTR\_HANDLE\_MAX (C macro), 135
- ESP\_GATT\_AUTH\_FAIL (C++ enumerator), 137
- ESP\_GATT\_AUTH\_REQ\_MITM (C++ enumerator), 138
- ESP\_GATT\_AUTH\_REQ\_NO\_MITM (C++ enumerator), 138
- ESP\_GATT\_AUTH\_REQ\_NONE (C++ enumerator), 138
- ESP\_GATT\_AUTH\_REQ\_SIGNED\_MITM (C++ enumerator), 138
- ESP\_GATT\_AUTH\_REQ\_SIGNED\_NO\_MITM (C++ enumerator), 138
- esp\_gatt\_auth\_req\_t (C++ type), 138
- ESP\_GATT\_AUTO\_RSP (C macro), 136
- ESP\_GATT\_BODY\_SENSOR\_LOCATION (C macro), 135
- ESP\_GATT\_BUSY (C++ enumerator), 137
- ESP\_GATT\_CANCEL (C++ enumerator), 138
- ESP\_GATT\_CCC\_CFG\_ERR (C++ enumerator), 138
- ESP\_GATT\_CHAR\_PROP\_BIT\_AUTH (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_BROADCAST (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_EXT\_PROP (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_INDICATE (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_NOTIFY (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_READ (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_WRITE (C macro), 136
- ESP\_GATT\_CHAR\_PROP\_BIT\_WRITE\_NR (C macro), 136
- esp\_gatt\_char\_prop\_t (C++ type), 136
- ESP\_GATT\_CMD\_STARTED (C++ enumerator), 137
- ESP\_GATT\_CONGESTED (C++ enumerator), 138
- ESP\_GATT\_CONN\_CONN\_CANCEL (C++ enumerator), 138
- ESP\_GATT\_CONN\_FAIL\_ESTABLISH (C++ enumerator), 138
- ESP\_GATT\_CONN\_L2C\_FAILURE (C++ enumerator), 138
- ESP\_GATT\_CONN\_LMP\_TIMEOUT (C++ enumerator), 138
- ESP\_GATT\_CONN\_NONE (C++ enumerator), 138
- esp\_gatt\_conn\_reason\_t (C++ type), 138
- ESP\_GATT\_CONN\_TERMINATE\_LOCAL\_HOST (C++ enumerator), 138
- ESP\_GATT\_CONN\_TERMINATE\_PEER\_USER (C++ enumerator), 138
- ESP\_GATT\_CONN\_TIMEOUT (C++ enumerator), 138
- ESP\_GATT\_CONN\_UNKNOWN (C++ enumerator), 138
- ESP\_GATT\_DB\_ALL (C++ enumerator), 139
- esp\_gatt\_db\_attr\_type\_t (C++ type), 139
- ESP\_GATT\_DB\_CHARACTERISTIC (C++ enumerator), 139
- ESP\_GATT\_DB\_DESCRIPTOR (C++ enumerator), 139
- ESP\_GATT\_DB\_FULL (C++ enumerator), 137
- ESP\_GATT\_DB\_INCLUDED\_SERVICE (C++ enumerator), 139
- ESP\_GATT\_DB\_PRIMARY\_SERVICE (C++ enumerator), 139
- ESP\_GATT\_DB\_SECONDARY\_SERVICE (C++ enumerator), 139
- ESP\_GATT\_DUP\_REG (C++ enumerator), 138
- ESP\_GATT\_ENCRYPED\_MITM (C++ enumerator), 137
- ESP\_GATT\_ENCRYPED\_NO\_MITM (C++ enumerator), 137
- ESP\_GATT\_ERR\_UNLIKELY (C++ enumerator), 137
- ESP\_GATT\_ERROR (C++ enumerator), 137
- ESP\_GATT\_HEART\_RATE\_CNTL\_POINT (C macro), 135
- ESP\_GATT\_HEART\_RATE\_MEAS (C macro), 135
- esp\_gatt\_id\_t (C++ class), 129
- esp\_gatt\_id\_t::inst\_id (C++ member), 129
- esp\_gatt\_id\_t::uuid (C++ member), 129
- ESP\_GATT\_IF\_NONE (C macro), 136
- esp\_gatt\_if\_t (C++ type), 136
- ESP\_GATT\_ILLEGAL\_HANDLE (C macro), 135
- ESP\_GATT\_ILLEGAL\_PARAMETER (C++ enumerator), 137
- ESP\_GATT\_ILLEGAL\_UUID (C macro), 135
- ESP\_GATT\_INSUF\_AUTHENTICATION (C++ enumerator), 137
- ESP\_GATT\_INSUF\_AUTHORIZATION (C++ enumerator), 137
- ESP\_GATT\_INSUF\_ENCRYPTION (C++ enumerator), 137
- ESP\_GATT\_INSUF\_KEY\_SIZE (C++ enumerator), 137
- ESP\_GATT\_INSUF\_RESOURCE (C++ enumerator), 137
- ESP\_GATT\_INTERNAL\_ERROR (C++ enumerator), 137
- ESP\_GATT\_INVALID\_ATTR\_LEN (C++ enumerator), 137
- ESP\_GATT\_INVALID\_CFG (C++ enumerator), 137
- ESP\_GATT\_INVALID\_HANDLE (C++ enumerator), 137
- ESP\_GATT\_INVALID\_OFFSET (C++ enumerator), 137
- ESP\_GATT\_INVALID\_PDU (C++ enumerator), 137

- ESP\_GATT\_MAX\_ATTR\_LEN (*C macro*), 136
- ESP\_GATT\_MAX\_READ\_MULTI\_HANDLES (*C macro*), 135
- ESP\_GATT\_MORE (*C++ enumerator*), 137
- ESP\_GATT\_NO\_RESOURCES (*C++ enumerator*), 137
- ESP\_GATT\_NOT\_ENCRYPTED (*C++ enumerator*), 137
- ESP\_GATT\_NOT\_FOUND (*C++ enumerator*), 137
- ESP\_GATT\_NOT\_LONG (*C++ enumerator*), 137
- ESP\_GATT\_OK (*C++ enumerator*), 137
- ESP\_GATT\_OUT\_OF\_RANGE (*C++ enumerator*), 138
- ESP\_GATT\_PENDING (*C++ enumerator*), 137
- ESP\_GATT\_PERM\_READ (*C macro*), 135
- ESP\_GATT\_PERM\_READ\_ENC\_MITM (*C macro*), 136
- ESP\_GATT\_PERM\_READ\_ENCRYPTED (*C macro*), 136
- esp\_gatt\_perm\_t (*C++ type*), 136
- ESP\_GATT\_PERM\_WRITE (*C macro*), 136
- ESP\_GATT\_PERM\_WRITE\_ENC\_MITM (*C macro*), 136
- ESP\_GATT\_PERM\_WRITE\_ENCRYPTED (*C macro*), 136
- ESP\_GATT\_PERM\_WRITE\_SIGNED (*C macro*), 136
- ESP\_GATT\_PERM\_WRITE\_SIGNED\_MITM (*C macro*), 136
- ESP\_GATT\_PRC\_IN\_PROGRESS (*C++ enumerator*), 138
- ESP\_GATT\_PREP\_WRITE\_CANCEL (*C macro*), 151
- ESP\_GATT\_PREP\_WRITE\_CANCEL (*C++ enumerator*), 136
- ESP\_GATT\_PREP\_WRITE\_EXEC (*C macro*), 151
- ESP\_GATT\_PREP\_WRITE\_EXEC (*C++ enumerator*), 136
- esp\_gatt\_prep\_write\_type (*C++ type*), 136
- ESP\_GATT\_PREPARE\_Q\_FULL (*C++ enumerator*), 137
- ESP\_GATT\_READ\_NOT\_PERMIT (*C++ enumerator*), 137
- ESP\_GATT\_REQ\_NOT\_SUPPORTED (*C++ enumerator*), 137
- ESP\_GATT\_RSP\_BY\_APP (*C macro*), 136
- esp\_gatt\_rsp\_t (*C++ type*), 129
- esp\_gatt\_rsp\_t::attr\_value (*C++ member*), 129
- esp\_gatt\_rsp\_t::handle (*C++ member*), 129
- ESP\_GATT\_SERVICE\_STARTED (*C++ enumerator*), 137
- esp\_gatt\_srvc\_id\_t (*C++ class*), 129
- esp\_gatt\_srvc\_id\_t::id (*C++ member*), 129
- esp\_gatt\_srvc\_id\_t::is\_primary (*C++ member*), 129
- ESP\_GATT\_STACK\_RSP (*C++ enumerator*), 138
- esp\_gatt\_status\_t (*C++ type*), 136
- ESP\_GATT\_UNKNOWN\_ERROR (*C++ enumerator*), 138
- ESP\_GATT\_UNSUPPORT\_GRP\_TYPE (*C++ enumerator*), 137
- ESP\_GATT\_UUID\_ALERT\_LEVEL (*C macro*), 134
- ESP\_GATT\_UUID\_ALERT\_NTF\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_ALERT\_STATUS (*C macro*), 134
- ESP\_GATT\_UUID\_BATTERY\_LEVEL (*C macro*), 135
- ESP\_GATT\_UUID\_BATTERY\_SERVICE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_BLOOD\_PRESSURE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_CHAR\_AGG\_FORMAT (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_CLIENT\_CONFIG (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_DECLARE (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_DESCRIPTION (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_EXT\_PROP (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_PRESENT\_FORMAT (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_SVR\_CONFIG (*C macro*), 134
- ESP\_GATT\_UUID\_CHAR\_VALID\_RANGE (*C macro*), 134
- ESP\_GATT\_UUID\_CSC\_FEATURE (*C macro*), 135
- ESP\_GATT\_UUID\_CSC\_MEASUREMENT (*C macro*), 135
- ESP\_GATT\_UUID\_CURRENT\_TIME (*C macro*), 134
- ESP\_GATT\_UUID\_CURRENT\_TIME\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_CYCLING\_POWER\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_CYCLING\_SPEED\_CADENCE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_DEVICE\_INFO\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_EXT\_RPT\_REF\_DESCR (*C macro*), 134
- ESP\_GATT\_UUID\_FW\_VERSION\_STR (*C macro*), 135
- ESP\_GATT\_UUID\_GAP\_CENTRAL\_ADDR\_RESOL (*C macro*), 134
- ESP\_GATT\_UUID\_GAP\_DEVICE\_NAME (*C macro*), 134
- ESP\_GATT\_UUID\_GAP\_ICON (*C macro*), 134
- ESP\_GATT\_UUID\_GAP\_PREF\_CONN\_PARAM (*C macro*), 134
- ESP\_GATT\_UUID\_GATT\_SRV\_CHGD (*C macro*), 134
- ESP\_GATT\_UUID\_GLUCOSE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_GM\_CONTEXT (*C macro*), 134
- ESP\_GATT\_UUID\_GM\_CONTROL\_POINT (*C macro*), 134
- ESP\_GATT\_UUID\_GM\_FEATURE (*C macro*), 134
- ESP\_GATT\_UUID\_GM\_MEASUREMENT (*C macro*), 134
- ESP\_GATT\_UUID\_HEALTH\_THERMOM\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_HEART\_RATE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_HID\_BT\_KB\_INPUT (*C macro*),

- 135
- ESP\_GATT\_UUID\_HID\_BT\_KB\_OUTPUT (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_BT\_MOUSE\_INPUT (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_CONTROL\_POINT (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_INFORMATION (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_PROTO\_MODE (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_REPORT (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_REPORT\_MAP (*C macro*), 135
- ESP\_GATT\_UUID\_HID\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_HW\_VERSION\_STR (*C macro*), 135
- ESP\_GATT\_UUID\_IEEE\_DATA (*C macro*), 135
- ESP\_GATT\_UUID\_IMMEDIATE\_ALERT\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_INCLUDE\_SERVICE (*C macro*), 134
- ESP\_GATT\_UUID\_LINK\_LOSS\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_LOCAL\_TIME\_INFO (*C macro*), 134
- ESP\_GATT\_UUID\_LOCATION\_AND\_NAVIGATION\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_MANU\_NAME (*C macro*), 135
- ESP\_GATT\_UUID\_MODEL\_NUMBER\_STR (*C macro*), 134
- ESP\_GATT\_UUID\_NEXT\_DST\_CHANGE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_NW\_STATUS (*C macro*), 134
- ESP\_GATT\_UUID\_NW\_TRIGGER (*C macro*), 134
- ESP\_GATT\_UUID\_PHONE\_ALERT\_STATUS\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_PNP\_ID (*C macro*), 135
- ESP\_GATT\_UUID\_PRI\_SERVICE (*C macro*), 134
- ESP\_GATT\_UUID\_REF\_TIME\_INFO (*C macro*), 134
- ESP\_GATT\_UUID\_REF\_TIME\_UPDATE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_RINGER\_CP (*C macro*), 134
- ESP\_GATT\_UUID\_RINGER\_SETTING (*C macro*), 134
- ESP\_GATT\_UUID\_RPT\_REF\_DESCR (*C macro*), 134
- ESP\_GATT\_UUID\_RSC\_FEATURE (*C macro*), 135
- ESP\_GATT\_UUID\_RSC\_MEASUREMENT (*C macro*), 135
- ESP\_GATT\_UUID\_RUNNING\_SPEED\_CADENCE\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_SC\_CONTROL\_POINT (*C macro*), 135
- ESP\_GATT\_UUID\_SCAN\_INT\_WINDOW (*C macro*), 135
- ESP\_GATT\_UUID\_SCAN\_PARAMETERS\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_SCAN\_REFRESH (*C macro*), 135
- ESP\_GATT\_UUID\_SEC\_SERVICE (*C macro*), 134
- ESP\_GATT\_UUID\_SENSOR\_LOCATION (*C macro*), 135
- ESP\_GATT\_UUID\_SERIAL\_NUMBER\_STR (*C macro*), 134
- ESP\_GATT\_UUID\_SW\_VERSION\_STR (*C macro*), 135
- ESP\_GATT\_UUID\_SYSTEM\_ID (*C macro*), 134
- ESP\_GATT\_UUID\_TX\_POWER\_LEVEL (*C macro*), 134
- ESP\_GATT\_UUID\_TX\_POWER\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_USER\_DATA\_SVC (*C macro*), 133
- ESP\_GATT\_UUID\_WEIGHT\_SCALE\_SVC (*C macro*), 134
- esp\_gatt\_value\_t (*C++ class*), 131
- esp\_gatt\_value\_t::auth\_req (*C++ member*), 131
- esp\_gatt\_value\_t::handle (*C++ member*), 131
- esp\_gatt\_value\_t::len (*C++ member*), 131
- esp\_gatt\_value\_t::offset (*C++ member*), 131
- esp\_gatt\_value\_t::value (*C++ member*), 131
- ESP\_GATT\_WRITE\_NOT\_PERMIT (*C++ enumerator*), 137
- ESP\_GATT\_WRITE\_TYPE\_NO\_RSP (*C++ enumerator*), 139
- ESP\_GATT\_WRITE\_TYPE\_RSP (*C++ enumerator*), 139
- esp\_gatt\_write\_type\_t (*C++ type*), 138
- ESP\_GATT\_WRONG\_STATE (*C++ enumerator*), 137
- ESP\_GATTC\_ACL\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_ADV\_DATA\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_ADV\_VSC\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_BTH\_SCAN\_CFG\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_BTH\_SCAN\_DIS\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_BTH\_SCAN\_ENB\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_BTH\_SCAN\_PARAM\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_BTH\_SCAN\_RD\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_BTH\_SCAN\_THR\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_CANCEL\_OPEN\_EVT (*C++ enumerator*), 169
- esp\_gattc\_cb\_event\_t (*C++ type*), 168
- esp\_gattc\_cb\_t (*C++ type*), 168
- ESP\_GATTC\_CFG\_MTU\_EVT (*C++ enumerator*), 169
- esp\_gattc\_char\_elem\_t (*C++ class*), 132
- esp\_gattc\_char\_elem\_t::char\_handle (*C++ member*), 132
- esp\_gattc\_char\_elem\_t::properties (*C++ member*), 132
- esp\_gattc\_char\_elem\_t::uuid (*C++ member*), 132
- ESP\_GATTC\_CLOSE\_EVT (*C++ enumerator*), 168

- ESP\_GATTC\_CONGEST\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_CONNECT\_EVT (*C++ enumerator*), 170
- esp\_gattc\_db\_elem\_t (*C++ class*), 131
- esp\_gattc\_db\_elem\_t::attribute\_handle (*C++ member*), 132
- esp\_gattc\_db\_elem\_t::end\_handle (*C++ member*), 132
- esp\_gattc\_db\_elem\_t::properties (*C++ member*), 132
- esp\_gattc\_db\_elem\_t::start\_handle (*C++ member*), 132
- esp\_gattc\_db\_elem\_t::type (*C++ member*), 132
- esp\_gattc\_db\_elem\_t::uuid (*C++ member*), 132
- esp\_gattc\_descr\_elem\_t (*C++ class*), 132
- esp\_gattc\_descr\_elem\_t::handle (*C++ member*), 133
- esp\_gattc\_descr\_elem\_t::uuid (*C++ member*), 133
- ESP\_GATTC\_DISCONNECT\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_ENC\_CMPL\_CB\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_EXEC\_EVT (*C++ enumerator*), 169
- esp\_gattc\_incl\_svc\_elem\_t (*C++ class*), 133
- esp\_gattc\_incl\_svc\_elem\_t::handle (*C++ member*), 133
- esp\_gattc\_incl\_svc\_elem\_t::incl\_srvc\_s\_handle (*C++ member*), 133
- esp\_gattc\_incl\_svc\_elem\_t::uuid (*C++ member*), 133
- ESP\_GATTC\_MULT\_ADV\_DATA\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_MULT\_ADV\_DIS\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_MULT\_ADV\_ENB\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_MULT\_ADV\_UPD\_EVT (*C++ enumerator*), 169
- esp\_gattc\_multi\_t (*C++ class*), 131
- esp\_gattc\_multi\_t::handles (*C++ member*), 131
- esp\_gattc\_multi\_t::num\_attr (*C++ member*), 131
- ESP\_GATTC\_NOTIFY\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_OPEN\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_PREP\_WRITE\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_QUEUE\_FULL\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_READ\_CHAR\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_READ\_DESCR\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_READ\_MUTIPLE\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_REG\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_REG\_FOR\_NOTIFY\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_SCAN\_FLT\_CFG\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_SCAN\_FLT\_PARAM\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_SCAN\_FLT\_STATUS\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_SEARCH\_CMPL\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_SEARCH\_RES\_EVT (*C++ enumerator*), 168
- esp\_gattc\_service\_elem\_t (*C++ class*), 132
- esp\_gattc\_service\_elem\_t::end\_handle (*C++ member*), 132
- esp\_gattc\_service\_elem\_t::is\_primary (*C++ member*), 132
- esp\_gattc\_service\_elem\_t::start\_handle (*C++ member*), 132
- esp\_gattc\_service\_elem\_t::uuid (*C++ member*), 132
- ESP\_GATTC\_SRVC\_CHG\_EVT (*C++ enumerator*), 169
- ESP\_GATTC\_UNREG\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_UNREG\_FOR\_NOTIFY\_EVT (*C++ enumerator*), 170
- ESP\_GATTC\_WRITE\_CHAR\_EVT (*C++ enumerator*), 168
- ESP\_GATTC\_WRITE\_DESCR\_EVT (*C++ enumerator*), 169
- ESP\_GATTS\_ADD\_CHAR\_DESCR\_EVT (*C++ enumerator*), 152
- ESP\_GATTS\_ADD\_CHAR\_EVT (*C++ enumerator*), 152
- ESP\_GATTS\_ADD\_INCL\_SRVC\_EVT (*C++ enumerator*), 152
- esp\_gatts\_attr\_db\_t (*C++ class*), 130
- esp\_gatts\_attr\_db\_t::att\_desc (*C++ member*), 130
- esp\_gatts\_attr\_db\_t::attr\_control (*C++ member*), 130
- ESP\_GATTS\_CANCEL\_OPEN\_EVT (*C++ enumerator*), 152
- esp\_gatts\_cb\_event\_t (*C++ type*), 151
- esp\_gatts\_cb\_t (*C++ type*), 151
- ESP\_GATTS\_CLOSE\_EVT (*C++ enumerator*), 152
- ESP\_GATTS\_CONF\_EVT (*C++ enumerator*), 152
- ESP\_GATTS\_CONGEST\_EVT (*C++ enumerator*), 152
- ESP\_GATTS\_CONNECT\_EVT (*C++ enumerator*), 152
- ESP\_GATTS\_CREAT\_ATTR\_TAB\_EVT (*C++ enumerator*), 153
- ESP\_GATTS\_CREATE\_EVT (*C++ enumerator*), 152

- ESP\_GATTS\_DELETE\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_DISCONNECT\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_EXEC\_WRITE\_EVT (C++ *enumerator*), 152
- esp\_gatts\_incl128\_svc\_desc\_t (C++ *class*), 131
- esp\_gatts\_incl128\_svc\_desc\_t::end\_hdl (C++ *member*), 131
- esp\_gatts\_incl128\_svc\_desc\_t::start\_hdl (C++ *member*), 131
- esp\_gatts\_incl\_svc\_desc\_t (C++ *class*), 130
- esp\_gatts\_incl\_svc\_desc\_t::end\_hdl (C++ *member*), 131
- esp\_gatts\_incl\_svc\_desc\_t::start\_hdl (C++ *member*), 131
- esp\_gatts\_incl\_svc\_desc\_t::uuid (C++ *member*), 131
- ESP\_GATTS\_LISTEN\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_MTU\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_OPEN\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_READ\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_REG\_EVT (C++ *enumerator*), 151
- ESP\_GATTS\_RESPONSE\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_SET\_ATTR\_VAL\_EVT (C++ *enumerator*), 153
- ESP\_GATTS\_START\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_STOP\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_UNREG\_EVT (C++ *enumerator*), 152
- ESP\_GATTS\_WRITE\_EVT (C++ *enumerator*), 152
- esp\_gcov\_dump (C++ *function*), 592
- esp\_int\_wdt\_init (C++ *function*), 574
- esp\_intr\_alloc (C++ *function*), 569
- esp\_intr\_alloc\_intrstatus (C++ *function*), 569
- esp\_intr\_disable (C++ *function*), 570
- esp\_intr\_enable (C++ *function*), 571
- ESP\_INTR\_FLAG\_EDGE (C *macro*), 571
- ESP\_INTR\_FLAG\_HIGH (C *macro*), 572
- ESP\_INTR\_FLAG\_INTRDISABLED (C *macro*), 571
- ESP\_INTR\_FLAG\_IRAM (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVEL1 (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVEL2 (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVEL3 (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVEL4 (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVEL5 (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVEL6 (C *macro*), 571
- ESP\_INTR\_FLAG\_LEVELMASK (C *macro*), 572
- ESP\_INTR\_FLAG\_LOWMED (C *macro*), 571
- ESP\_INTR\_FLAG\_NMI (C *macro*), 571
- ESP\_INTR\_FLAG\_SHARED (C *macro*), 571
- esp\_intr\_free (C++ *function*), 570
- esp\_intr\_get\_cpu (C++ *function*), 570
- esp\_intr\_get\_intno (C++ *function*), 570
- esp\_intr\_mark\_shared (C++ *function*), 568
- esp\_intr\_noniram\_disable (C++ *function*), 571
- esp\_intr\_noniram\_enable (C++ *function*), 571
- esp\_intr\_reserve (C++ *function*), 568
- ESP\_IO\_CAP\_IN (C *macro*), 123
- ESP\_IO\_CAP\_IO (C *macro*), 123
- ESP\_IO\_CAP\_KBDISP (C *macro*), 123
- ESP\_IO\_CAP\_NONE (C *macro*), 123
- ESP\_IO\_CAP\_OUT (C *macro*), 123
- esp\_ipc\_call (C++ *function*), 577
- esp\_ipc\_call\_blocking (C++ *function*), 577
- ESP\_LE\_AUTH\_BOND (C *macro*), 122
- ESP\_LE\_AUTH\_NO\_BOND (C *macro*), 122
- ESP\_LE\_AUTH\_REQ\_MITM (C *macro*), 122
- ESP\_LE\_AUTH\_REQ\_SC\_BOND (C *macro*), 123
- ESP\_LE\_AUTH\_REQ\_SC\_MITM (C *macro*), 123
- ESP\_LE\_AUTH\_REQ\_SC\_MITM\_BOND (C *macro*), 123
- ESP\_LE\_AUTH\_REQ\_SC\_ONLY (C *macro*), 122
- ESP\_LE\_KEY\_LCSRK (C *macro*), 122
- ESP\_LE\_KEY\_LENC (C *macro*), 122
- ESP\_LE\_KEY\_LID (C *macro*), 122
- ESP\_LE\_KEY\_LLK (C *macro*), 122
- ESP\_LE\_KEY\_NONE (C *macro*), 122
- ESP\_LE\_KEY\_PCSRK (C *macro*), 122
- ESP\_LE\_KEY\_PENC (C *macro*), 122
- ESP\_LE\_KEY\_PID (C *macro*), 122
- ESP\_LE\_KEY\_PLK (C *macro*), 122
- esp\_light\_sleep\_start (C++ *function*), 603
- ESP\_LINE\_ENDINGS\_CR (C++ *enumerator*), 438
- ESP\_LINE\_ENDINGS\_CRLF (C++ *enumerator*), 437
- ESP\_LINE\_ENDINGS\_LF (C++ *enumerator*), 438
- esp\_line\_endings\_t (C++ *type*), 437
- esp\_link\_key (C++ *type*), 101
- ESP\_LOG\_BUFFER\_CHAR (C *macro*), 585
- esp\_log\_buffer\_char (C *macro*), 586
- ESP\_LOG\_BUFFER\_CHAR\_LEVEL (C *macro*), 585
- ESP\_LOG\_BUFFER\_HEX (C *macro*), 585
- esp\_log\_buffer\_hex (C *macro*), 586
- ESP\_LOG\_BUFFER\_HEX\_LEVEL (C *macro*), 584
- ESP\_LOG\_BUFFER\_HEXDUMP (C *macro*), 585
- ESP\_LOG\_DEBUG (C++ *enumerator*), 587
- esp\_log\_early\_timestamp (C++ *function*), 584
- ESP\_LOG\_ERROR (C++ *enumerator*), 587
- ESP\_LOG\_INFO (C++ *enumerator*), 587
- ESP\_LOG\_LEVEL (C *macro*), 587
- ESP\_LOG\_LEVEL\_LOCAL (C *macro*), 587
- esp\_log\_level\_set (C++ *function*), 584
- esp\_log\_level\_t (C++ *type*), 587
- ESP\_LOG\_NONE (C++ *enumerator*), 587
- esp\_log\_set\_vprintf (C++ *function*), 584
- esp\_log\_timestamp (C++ *function*), 584
- ESP\_LOG\_VERBOSE (C++ *enumerator*), 587
- ESP\_LOG\_WARN (C++ *enumerator*), 587
- esp\_log\_write (C++ *function*), 584

- ESP\_LOGD (C macro), 586
- ESP\_LOGE (C macro), 586
- ESP\_LOGI (C macro), 586
- ESP\_LOGV (C macro), 587
- ESP\_LOGW (C macro), 586
- esp\_now\_add\_peer (C++ function), 94
- esp\_now\_deinit (C++ function), 92
- esp\_now\_del\_peer (C++ function), 94
- ESP\_NOW\_ETH\_ALEN (C macro), 97
- esp\_now\_fetch\_peer (C++ function), 95
- esp\_now\_get\_peer (C++ function), 95
- esp\_now\_get\_peer\_num (C++ function), 96
- esp\_now\_get\_version (C++ function), 92
- esp\_now\_init (C++ function), 92
- esp\_now\_is\_peer\_exist (C++ function), 95
- ESP\_NOW\_KEY\_LEN (C macro), 97
- ESP\_NOW\_MAX\_DATA\_LEN (C macro), 97
- ESP\_NOW\_MAX\_ENCRYPT\_PEER\_NUM (C macro), 97
- ESP\_NOW\_MAX\_TOTAL\_PEER\_NUM (C macro), 97
- esp\_now\_mod\_peer (C++ function), 94
- esp\_now\_peer\_info (C++ class), 96
- esp\_now\_peer\_info::channel (C++ member), 96
- esp\_now\_peer\_info::encrypt (C++ member), 96
- esp\_now\_peer\_info::ifidx (C++ member), 96
- esp\_now\_peer\_info::lmk (C++ member), 96
- esp\_now\_peer\_info::peer\_addr (C++ member), 96
- esp\_now\_peer\_info::priv (C++ member), 96
- esp\_now\_peer\_info\_t (C++ type), 98
- esp\_now\_peer\_num (C++ class), 96
- esp\_now\_peer\_num::encrypt\_num (C++ member), 97
- esp\_now\_peer\_num::total\_num (C++ member), 97
- esp\_now\_peer\_num\_t (C++ type), 98
- esp\_now\_rcv\_cb\_t (C++ type), 98
- esp\_now\_register\_rcv\_cb (C++ function), 93
- esp\_now\_register\_send\_cb (C++ function), 93
- esp\_now\_send (C++ function), 93
- esp\_now\_send\_cb\_t (C++ type), 98
- ESP\_NOW\_SEND\_FAIL (C++ enumerator), 98
- esp\_now\_send\_status\_t (C++ type), 98
- ESP\_NOW\_SEND\_SUCCESS (C++ enumerator), 98
- esp\_now\_set\_pmk (C++ function), 96
- esp\_now\_unregister\_rcv\_cb (C++ function), 93
- esp\_now\_unregister\_send\_cb (C++ function), 93
- esp\_ota\_begin (C++ function), 607
- esp\_ota\_end (C++ function), 608
- esp\_ota\_get\_boot\_partition (C++ function), 609
- esp\_ota\_get\_next\_update\_partition (C++ function), 609
- esp\_ota\_get\_running\_partition (C++ function), 609
- esp\_ota\_handle\_t (C++ type), 610
- esp\_ota\_set\_boot\_partition (C++ function), 608
- esp\_ota\_write (C++ function), 608
- esp\_partition\_erase\_range (C++ function), 413
- esp\_partition\_find (C++ function), 411
- esp\_partition\_find\_first (C++ function), 411
- esp\_partition\_get (C++ function), 412
- esp\_partition\_iterator\_release (C++ function), 412
- esp\_partition\_iterator\_t (C++ type), 415
- esp\_partition\_mmap (C++ function), 414
- esp\_partition\_next (C++ function), 412
- esp\_partition\_read (C++ function), 413
- ESP\_PARTITION\_SUBTYPE\_ANY (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_FACTORY (C++ enumerator), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_0 (C++ enumerator), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_1 (C++ enumerator), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_10 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_11 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_12 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_13 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_14 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_15 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_2 (C++ enumerator), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_3 (C++ enumerator), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_4 (C++ enumerator), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_5 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_6 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_7 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_8 (C++ enumerator), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_9 (C++ enumerator), 416

- enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MAX (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_APP\_OTA\_MIN (C++ *enumerator*), 415
- ESP\_PARTITION\_SUBTYPE\_APP\_TEST (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_COREDUMP (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_ESPHTTPD (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_FAT (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_NVS (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_OTA (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_PHY (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_DATA\_SPIFFS (C++ *enumerator*), 416
- ESP\_PARTITION\_SUBTYPE\_OTA (C *macro*), 415
- esp\_partition\_subtype\_t (C++ *type*), 415
- esp\_partition\_t (C++ *class*), 414
- esp\_partition\_t::address (C++ *member*), 414
- esp\_partition\_t::encrypted (C++ *member*), 415
- esp\_partition\_t::label (C++ *member*), 414
- esp\_partition\_t::size (C++ *member*), 414
- esp\_partition\_t::subtype (C++ *member*), 414
- esp\_partition\_t::type (C++ *member*), 414
- ESP\_PARTITION\_TYPE\_APP (C++ *enumerator*), 415
- ESP\_PARTITION\_TYPE\_DATA (C++ *enumerator*), 415
- esp\_partition\_type\_t (C++ *type*), 415
- esp\_partition\_verify (C++ *function*), 412
- esp\_partition\_write (C++ *function*), 413
- ESP\_PD\_DOMAIN\_MAX (C++ *enumerator*), 603
- ESP\_PD\_DOMAIN\_RTC\_FAST\_MEM (C++ *enumerator*), 602
- ESP\_PD\_DOMAIN\_RTC\_PERIPH (C++ *enumerator*), 602
- ESP\_PD\_DOMAIN\_RTC\_SLOW\_MEM (C++ *enumerator*), 602
- ESP\_PD\_OPTION\_AUTO (C++ *enumerator*), 603
- ESP\_PD\_OPTION\_OFF (C++ *enumerator*), 603
- ESP\_PD\_OPTION\_ON (C++ *enumerator*), 603
- ESP\_PM\_APB\_FREQ\_MAX (C++ *enumerator*), 597
- esp\_pm\_config\_esp32\_t (C++ *class*), 598
- esp\_pm\_config\_esp32\_t::light\_sleep\_enable (C++ *member*), 598
- esp\_pm\_config\_esp32\_t::max\_cpu\_freq (C++ *member*), 598
- esp\_pm\_config\_esp32\_t::min\_cpu\_freq (C++ *member*), 598
- esp\_pm\_configure (C++ *function*), 595
- ESP\_PM\_CPU\_FREQ\_MAX (C++ *enumerator*), 597
- esp\_pm\_dump\_locks (C++ *function*), 597
- esp\_pm\_lock\_acquire (C++ *function*), 596
- esp\_pm\_lock\_create (C++ *function*), 595
- esp\_pm\_lock\_delete (C++ *function*), 596
- esp\_pm\_lock\_handle\_t (C++ *type*), 597
- esp\_pm\_lock\_release (C++ *function*), 596
- esp\_pm\_lock\_type\_t (C++ *type*), 597
- ESP\_PM\_NO\_LIGHT\_SLEEP (C++ *enumerator*), 597
- esp\_register\_freertos\_idle\_hook (C++ *function*), 545
- esp\_register\_freertos\_idle\_hook\_for\_cpu (C++ *function*), 545
- esp\_register\_freertos\_tick\_hook (C++ *function*), 545
- esp\_register\_freertos\_tick\_hook\_for\_cpu (C++ *function*), 545
- esp\_sleep\_enable\_ext0\_wakeup (C++ *function*), 600
- esp\_sleep\_enable\_ext1\_wakeup (C++ *function*), 601
- esp\_sleep\_enable\_timer\_wakeup (C++ *function*), 599
- esp\_sleep\_enable\_touchpad\_wakeup (C++ *function*), 599
- esp\_sleep\_enable\_ulp\_wakeup (C++ *function*), 602
- esp\_sleep\_ext1\_wakeup\_mode\_t (C++ *type*), 601
- esp\_sleep\_get\_ext1\_wakeup\_status (C++ *function*), 604
- esp\_sleep\_get\_touchpad\_wakeup\_status (C++ *function*), 604
- esp\_sleep\_get\_wakeup\_cause (C++ *function*), 603
- esp\_sleep\_pd\_config (C++ *function*), 602
- esp\_sleep\_pd\_domain\_t (C++ *type*), 602
- esp\_sleep\_pd\_option\_t (C++ *type*), 603
- esp\_sleep\_wakeup\_cause\_t (C++ *type*), 603
- ESP\_SLEEP\_WAKEUP\_EXT0 (C++ *enumerator*), 604
- ESP\_SLEEP\_WAKEUP\_EXT1 (C++ *enumerator*), 604
- ESP\_SLEEP\_WAKEUP\_TIMER (C++ *enumerator*), 604
- ESP\_SLEEP\_WAKEUP\_TOUCHPAD (C++ *enumerator*), 604
- ESP\_SLEEP\_WAKEUP\_ULP (C++ *enumerator*), 604
- ESP\_SLEEP\_WAKEUP\_UNDEFINED (C++ *enumerator*), 604
- esp\_smartconfig\_fast\_mode (C++ *function*), 89
- esp\_smartconfig\_get\_version (C++ *function*), 88
- esp\_smartconfig\_set\_type (C++ *function*), 89
- esp\_smartconfig\_start (C++ *function*), 88

- esp\_smartconfig\_stop (C++ function), 88
- esp\_spiffs\_format (C++ function), 448
- esp\_spiffs\_info (C++ function), 448
- esp\_spiffs\_mounted (C++ function), 448
- esp\_task\_wdt\_add (C++ function), 574
- esp\_task\_wdt\_deinit (C++ function), 574
- esp\_task\_wdt\_delete (C++ function), 575
- esp\_task\_wdt\_feed (C++ function), 576
- esp\_task\_wdt\_init (C++ function), 574
- esp\_task\_wdt\_reset (C++ function), 575
- esp\_task\_wdt\_status (C++ function), 575
- esp\_timer\_cb\_t (C++ type), 582
- esp\_timer\_create (C++ function), 579
- esp\_timer\_create\_args\_t (C++ class), 581
- esp\_timer\_create\_args\_t::arg (C++ member), 581
- esp\_timer\_create\_args\_t::callback (C++ member), 581
- esp\_timer\_create\_args\_t::dispatch\_method (C++ member), 581
- esp\_timer\_create\_args\_t::name (C++ member), 581
- esp\_timer\_deinit (C++ function), 579
- esp\_timer\_delete (C++ function), 580
- esp\_timer\_dispatch\_t (C++ type), 582
- esp\_timer\_dump (C++ function), 581
- esp\_timer\_get\_time (C++ function), 581
- esp\_timer\_handle\_t (C++ type), 582
- esp\_timer\_init (C++ function), 579
- esp\_timer\_start\_once (C++ function), 579
- esp\_timer\_start\_periodic (C++ function), 580
- esp\_timer\_stop (C++ function), 580
- ESP\_TIMER\_TASK (C++ enumerator), 582
- ESP\_UUID\_LEN\_128 (C macro), 100
- ESP\_UUID\_LEN\_16 (C macro), 100
- ESP\_UUID\_LEN\_32 (C macro), 100
- esp\_vendor\_ie\_cb\_t (C++ type), 77
- esp\_vfs\_close (C++ function), 434
- esp\_vfs\_dev\_uart\_register (C++ function), 436
- esp\_vfs\_dev\_uart\_set\_rx\_line\_endings (C++ function), 436
- esp\_vfs\_dev\_uart\_set\_tx\_line\_endings (C++ function), 437
- esp\_vfs\_dev\_uart\_use\_driver (C++ function), 437
- esp\_vfs\_dev\_uart\_use\_nonblocking (C++ function), 437
- esp\_vfs\_fat\_mount\_config\_t (C++ class), 440, 443
- esp\_vfs\_fat\_mount\_config\_t::format\_if\_mounted (C++ member), 440, 443
- esp\_vfs\_fat\_mount\_config\_t::max\_files (C++ member), 440, 443
- esp\_vfs\_fat\_register (C++ function), 438
- esp\_vfs\_fat\_sdmmc\_mount (C++ function), 439
- esp\_vfs\_fat\_sdmmc\_unmount (C++ function), 440
- esp\_vfs\_fat\_spiflash\_mount (C++ function), 443
- esp\_vfs\_fat\_spiflash\_unmount (C++ function), 443
- esp\_vfs\_fat\_unregister\_path (C++ function), 439
- ESP\_VFS\_FLAG\_CONTEXT\_PTR (C macro), 436
- ESP\_VFS\_FLAG\_DEFAULT (C macro), 436
- ESP\_VFS\_FLAG\_SHARED\_FD\_SPACE (C macro), 436
- esp\_vfs\_fstat (C++ function), 435
- esp\_vfs\_link (C++ function), 435
- esp\_vfs\_lseek (C++ function), 434
- esp\_vfs\_open (C++ function), 434
- ESP\_VFS\_PATH\_MAX (C macro), 436
- esp\_vfs\_read (C++ function), 434
- esp\_vfs\_register (C++ function), 435
- esp\_vfs\_register\_socket\_space (C++ function), 435
- esp\_vfs\_rename (C++ function), 435
- esp\_vfs\_spiffs\_conf\_t (C++ class), 449
- esp\_vfs\_spiffs\_conf\_t::base\_path (C++ member), 449
- esp\_vfs\_spiffs\_conf\_t::format\_if\_mount\_failed (C++ member), 449
- esp\_vfs\_spiffs\_conf\_t::max\_files (C++ member), 449
- esp\_vfs\_spiffs\_conf\_t::partition\_label (C++ member), 449
- esp\_vfs\_spiffs\_register (C++ function), 447
- esp\_vfs\_spiffs\_unregister (C++ function), 447
- esp\_vfs\_stat (C++ function), 435
- esp\_vfs\_t (C++ class), 436
- esp\_vfs\_t::flags (C++ member), 436
- esp\_vfs\_unlink (C++ function), 435
- esp\_vfs\_unregister (C++ function), 435
- esp\_vfs\_write (C++ function), 434
- esp\_wifi\_ap\_get\_sta\_list (C++ function), 72
- esp\_wifi\_clear\_fast\_connect (C++ function), 64
- esp\_wifi\_connect (C++ function), 63
- esp\_wifi\_deauth\_sta (C++ function), 64
- esp\_wifi\_deinit (C++ function), 62
- esp\_wifi\_disconnect (C++ function), 64
- esp\_wifi\_get\_auto\_connect (C++ function), 73
- esp\_wifi\_get\_bandwidth (C++ function), 67
- esp\_wifi\_get\_channel (C++ function), 68
- esp\_wifi\_get\_config (C++ function), 72
- esp\_wifi\_get\_country (C++ function), 69
- esp\_wifi\_get\_mac (C++ function), 70

- esp\_wifi\_get\_max\_tx\_power (C++ function), 74
  - esp\_wifi\_get\_mode (C++ function), 62
  - esp\_wifi\_get\_promiscuous (C++ function), 70
  - esp\_wifi\_get\_promiscuous\_filter (C++ function), 71
  - esp\_wifi\_get\_protocol (C++ function), 67
  - esp\_wifi\_get\_ps (C++ function), 66
  - esp\_wifi\_init (C++ function), 62
  - ESP\_WIFI\_MAX\_CONN\_NUM (C macro), 84
  - esp\_wifi\_restore (C++ function), 63
  - esp\_wifi\_scan\_get\_ap\_num (C++ function), 65
  - esp\_wifi\_scan\_get\_ap\_records (C++ function), 65
  - esp\_wifi\_scan\_start (C++ function), 64
  - esp\_wifi\_scan\_stop (C++ function), 65
  - esp\_wifi\_set\_auto\_connect (C++ function), 72
  - esp\_wifi\_set\_bandwidth (C++ function), 67
  - esp\_wifi\_set\_channel (C++ function), 68
  - esp\_wifi\_set\_config (C++ function), 71
  - esp\_wifi\_set\_country (C++ function), 68
  - esp\_wifi\_set\_mac (C++ function), 69
  - esp\_wifi\_set\_max\_tx\_power (C++ function), 73
  - esp\_wifi\_set\_mode (C++ function), 62
  - esp\_wifi\_set\_promiscuous (C++ function), 70
  - esp\_wifi\_set\_promiscuous\_filter (C++ function), 71
  - esp\_wifi\_set\_promiscuous\_rx\_cb (C++ function), 70
  - esp\_wifi\_set\_protocol (C++ function), 66
  - esp\_wifi\_set\_ps (C++ function), 66
  - esp\_wifi\_set\_storage (C++ function), 72
  - esp\_wifi\_set\_vendor\_ie (C++ function), 73
  - esp\_wifi\_set\_vendor\_ie\_cb (C++ function), 73
  - esp\_wifi\_sta\_get\_ap\_info (C++ function), 66
  - esp\_wifi\_start (C++ function), 63
  - esp\_wifi\_stop (C++ function), 63
  - eStandardSleep (C++ enumerator), 474
  - eSuspended (C++ enumerator), 474
  - eTaskGetState (C++ function), 456
  - eTaskState (C++ type), 474
  - ETH\_CLOCK\_GPIO0\_IN (C++ enumerator), 198
  - ETH\_CLOCK\_GPIO0\_OUT (C++ enumerator), 198
  - ETH\_CLOCK\_GPIO16\_OUT (C++ enumerator), 198
  - ETH\_CLOCK\_GPIO17\_OUT (C++ enumerator), 198
  - eth\_clock\_mode\_t (C++ type), 198
  - eth\_config\_t (C++ class), 196
  - eth\_config\_t::clock\_mode (C++ member), 197
  - eth\_config\_t::flow\_ctrl\_enable (C++ member), 197
  - eth\_config\_t::gpio\_config (C++ member), 197
  - eth\_config\_t::mac\_mode (C++ member), 197
  - eth\_config\_t::phy\_addr (C++ member), 197
  - eth\_config\_t::phy\_check\_init (C++ member), 197
  - eth\_config\_t::phy\_check\_link (C++ member), 197
  - eth\_config\_t::phy\_get\_duplex\_mode (C++ member), 197
  - eth\_config\_t::phy\_get\_partner\_pause\_enable (C++ member), 197
  - eth\_config\_t::phy\_get\_speed\_mode (C++ member), 197
  - eth\_config\_t::phy\_init (C++ member), 197
  - eth\_config\_t::phy\_power\_enable (C++ member), 197
  - eth\_config\_t::tcpip\_input (C++ member), 197
  - eth\_duplex\_mode\_t (C++ type), 198
  - eth\_gpio\_config\_func (C++ type), 197
  - ETH\_MODE\_FULLDUPLEX (C++ enumerator), 198
  - ETH\_MODE\_HALFDUPLEX (C++ enumerator), 198
  - ETH\_MODE\_MII (C++ enumerator), 198
  - ETH\_MODE\_RMII (C++ enumerator), 198
  - eth\_mode\_t (C++ type), 198
  - eth\_phy\_base\_t (C++ type), 198
  - eth\_phy\_check\_init\_func (C++ type), 197
  - eth\_phy\_check\_link\_func (C++ type), 197
  - eth\_phy\_func (C++ type), 197
  - eth\_phy\_get\_duplex\_mode\_func (C++ type), 197
  - eth\_phy\_get\_partner\_pause\_enable\_func (C++ type), 197
  - eth\_phy\_get\_speed\_mode\_func (C++ type), 197
  - eth\_phy\_power\_enable\_func (C++ type), 197
  - ETH\_SPEED\_MODE\_100M (C++ enumerator), 198
  - ETH\_SPEED\_MODE\_10M (C++ enumerator), 198
  - eth\_speed\_mode\_t (C++ type), 198
  - eth\_tcpip\_input\_func (C++ type), 197
  - ETS\_INTERNAL\_INTR\_SOURCE\_OFF (C macro), 572
  - ETS\_INTERNAL\_PROFILING\_INTR\_SOURCE (C macro), 572
  - ETS\_INTERNAL\_SW0\_INTR\_SOURCE (C macro), 572
  - ETS\_INTERNAL\_SW1\_INTR\_SOURCE (C macro), 572
  - ETS\_INTERNAL\_TIMER0\_INTR\_SOURCE (C macro), 572
  - ETS\_INTERNAL\_TIMER1\_INTR\_SOURCE (C macro), 572
  - ETS\_INTERNAL\_TIMER2\_INTR\_SOURCE (C macro), 572
  - EventBits\_t (C++ type), 538
  - EventGroupHandle\_t (C++ type), 538
- ## F
- ff\_diskio\_impl\_t (C++ class), 441
  - ff\_diskio\_impl\_t::init (C++ member), 441
  - ff\_diskio\_impl\_t::ioctl (C++ member), 441

ff\_diskio\_impl\_t::read (C++ member), 441  
 ff\_diskio\_impl\_t::status (C++ member), 441  
 ff\_diskio\_impl\_t::write (C++ member), 441  
 ff\_diskio\_register (C++ function), 440  
 ff\_diskio\_register\_sdmmc (C++ function), 441

## G

gpio\_config (C++ function), 220  
 gpio\_config\_t (C++ class), 225  
 gpio\_config\_t::intr\_type (C++ member), 225  
 gpio\_config\_t::mode (C++ member), 225  
 gpio\_config\_t::pin\_bit\_mask (C++ member), 225  
 gpio\_config\_t::pull\_down\_en (C++ member), 225  
 gpio\_config\_t::pull\_up\_en (C++ member), 225  
 GPIO\_DRIVE\_CAP\_0 (C++ enumerator), 227  
 GPIO\_DRIVE\_CAP\_1 (C++ enumerator), 227  
 GPIO\_DRIVE\_CAP\_2 (C++ enumerator), 227  
 GPIO\_DRIVE\_CAP\_3 (C++ enumerator), 227  
 GPIO\_DRIVE\_CAP\_DEFAULT (C++ enumerator), 227  
 GPIO\_DRIVE\_CAP\_MAX (C++ enumerator), 227  
 gpio\_drive\_cap\_t (C++ type), 227  
 GPIO\_FLOATING (C++ enumerator), 227  
 gpio\_get\_drive\_capability (C++ function), 225  
 gpio\_get\_level (C++ function), 221  
 gpio\_install\_isr\_service (C++ function), 223  
 gpio\_int\_type\_t (C++ type), 226  
 GPIO\_INTR\_ANYEDGE (C++ enumerator), 226  
 GPIO\_INTR\_DISABLE (C++ enumerator), 226  
 gpio\_intr\_disable (C++ function), 220  
 gpio\_intr\_enable (C++ function), 220  
 GPIO\_INTR\_HIGH\_LEVEL (C++ enumerator), 226  
 GPIO\_INTR\_LOW\_LEVEL (C++ enumerator), 226  
 GPIO\_INTR\_MAX (C++ enumerator), 226  
 GPIO\_INTR\_NEGEDGE (C++ enumerator), 226  
 GPIO\_INTR\_POSEDGE (C++ enumerator), 226  
 GPIO\_IS\_VALID\_GPIO (C macro), 225  
 GPIO\_IS\_VALID\_OUTPUT\_GPIO (C macro), 225  
 gpio\_isr\_handle\_t (C++ type), 226  
 gpio\_isr\_handler\_add (C++ function), 224  
 gpio\_isr\_handler\_remove (C++ function), 224  
 gpio\_isr\_register (C++ function), 222  
 gpio\_isr\_t (C++ type), 226  
 GPIO\_MODE\_DISABLE (C++ enumerator), 226  
 GPIO\_MODE\_INPUT (C++ enumerator), 226  
 GPIO\_MODE\_INPUT\_OUTPUT (C++ enumerator), 227  
 GPIO\_MODE\_INPUT\_OUTPUT\_OD (C++ enumerator), 227  
 GPIO\_MODE\_OUTPUT (C++ enumerator), 226  
 GPIO\_MODE\_OUTPUT\_OD (C++ enumerator), 227  
 gpio\_mode\_t (C++ type), 226

GPIO\_NUM\_0 (C++ enumerator), 226  
 GPIO\_NUM\_1 (C++ enumerator), 226  
 GPIO\_NUM\_2 (C++ enumerator), 226  
 gpio\_num\_t (C++ type), 226  
 gpio\_pull\_mode\_t (C++ type), 227  
 gpio\_pulldown\_dis (C++ function), 223  
 GPIO\_PULLDOWN\_DISABLE (C++ enumerator), 227  
 gpio\_pulldown\_en (C++ function), 223  
 GPIO\_PULLDOWN\_ENABLE (C++ enumerator), 227  
 GPIO\_PULLDOWN\_ONLY (C++ enumerator), 227  
 gpio\_pulldown\_t (C++ type), 227  
 gpio\_pullup\_dis (C++ function), 223  
 GPIO\_PULLUP\_DISABLE (C++ enumerator), 227  
 gpio\_pullup\_en (C++ function), 222  
 GPIO\_PULLUP\_ENABLE (C++ enumerator), 227  
 GPIO\_PULLUP\_ONLY (C++ enumerator), 227  
 GPIO\_PULLUP\_PULLDOWN (C++ enumerator), 227  
 gpio\_pullup\_t (C++ type), 227  
 GPIO\_SEL\_0 (C macro), 225  
 GPIO\_SEL\_1 (C macro), 225  
 GPIO\_SEL\_2 (C macro), 225  
 gpio\_set\_direction (C++ function), 221  
 gpio\_set\_drive\_capability (C++ function), 224  
 gpio\_set\_intr\_type (C++ function), 220  
 gpio\_set\_level (C++ function), 221  
 gpio\_set\_pull\_mode (C++ function), 221  
 gpio\_uninstall\_isr\_service (C++ function), 224  
 gpio\_wakeup\_disable (C++ function), 222  
 gpio\_wakeup\_enable (C++ function), 222

## H

hall\_sensor\_read (C++ function), 208  
 heap\_caps\_add\_region (C++ function), 553  
 heap\_caps\_add\_region\_with\_caps (C++ function), 553  
 heap\_caps\_calloc (C++ function), 548  
 heap\_caps\_calloc\_prefer (C++ function), 551  
 heap\_caps\_check\_integrity (C++ function), 550  
 heap\_caps\_check\_integrity\_addr (C++ function), 550  
 heap\_caps\_check\_integrity\_all (C++ function), 549  
 heap\_caps\_dump (C++ function), 551  
 heap\_caps\_dump\_all (C++ function), 552  
 heap\_caps\_enable\_nonos\_stack\_heaps (C++ function), 553  
 heap\_caps\_free (C++ function), 548  
 heap\_caps\_get\_free\_size (C++ function), 548  
 heap\_caps\_get\_info (C++ function), 549  
 heap\_caps\_get\_largest\_free\_block (C++ function), 549

[heap\\_caps\\_get\\_minimum\\_free\\_size \(C++ function\), 549](#)  
[heap\\_caps\\_init \(C++ function\), 553](#)  
[heap\\_caps\\_malloc \(C++ function\), 547](#)  
[heap\\_caps\\_malloc\\_extmem\\_enable \(C++ function\), 550](#)  
[heap\\_caps\\_malloc\\_prefer \(C++ function\), 551](#)  
[heap\\_caps\\_print\\_heap\\_info \(C++ function\), 549](#)  
[heap\\_caps\\_realloc \(C++ function\), 548](#)  
[heap\\_caps\\_realloc\\_prefer \(C++ function\), 551](#)  
[HEAP\\_TRACE\\_ALL \(C++ enumerator\), 566](#)  
[heap\\_trace\\_dump \(C++ function\), 565](#)  
[heap\\_trace\\_get \(C++ function\), 565](#)  
[heap\\_trace\\_get\\_count \(C++ function\), 565](#)  
[heap\\_trace\\_init\\_standalone \(C++ function\), 564](#)  
[HEAP\\_TRACE\\_LEAKS \(C++ enumerator\), 566](#)  
[heap\\_trace\\_mode\\_t \(C++ type\), 566](#)  
[heap\\_trace\\_record\\_t \(C++ class\), 565](#)  
[heap\\_trace\\_record\\_t::address \(C++ member\), 565](#)  
[heap\\_trace\\_record\\_t::allocated\\_by \(C++ member\), 566](#)  
[heap\\_trace\\_record\\_t::ccount \(C++ member\), 565](#)  
[heap\\_trace\\_record\\_t::freed\\_by \(C++ member\), 566](#)  
[heap\\_trace\\_record\\_t::size \(C++ member\), 566](#)  
[heap\\_trace\\_resume \(C++ function\), 564](#)  
[heap\\_trace\\_start \(C++ function\), 564](#)  
[heap\\_trace\\_stop \(C++ function\), 564](#)  
[HSPI\\_HOST \(C++ enumerator\), 336](#)

I

[i2c\\_ack\\_type\\_t \(C++ type\), 248](#)  
[I2C\\_ADDR\\_BIT\\_10 \(C++ enumerator\), 248](#)  
[I2C\\_ADDR\\_BIT\\_7 \(C++ enumerator\), 248](#)  
[I2C\\_ADDR\\_BIT\\_MAX \(C++ enumerator\), 248](#)  
[i2c\\_addr\\_mode\\_t \(C++ type\), 248](#)  
[I2C\\_APB\\_CLK\\_FREQ \(C macro\), 247](#)  
[I2C\\_CMD\\_END \(C++ enumerator\), 248](#)  
[i2c\\_cmd\\_handle\\_t \(C++ type\), 247](#)  
[i2c\\_cmd\\_link\\_create \(C++ function\), 239](#)  
[i2c\\_cmd\\_link\\_delete \(C++ function\), 239](#)  
[I2C\\_CMD\\_READ \(C++ enumerator\), 248](#)  
[I2C\\_CMD\\_RESTART \(C++ enumerator\), 247](#)  
[I2C\\_CMD\\_STOP \(C++ enumerator\), 248](#)  
[I2C\\_CMD\\_WRITE \(C++ enumerator\), 248](#)  
[i2c\\_config\\_t \(C++ class\), 246](#)  
[i2c\\_config\\_t::addr\\_10bit\\_en \(C++ member\), 247](#)  
[i2c\\_config\\_t::clk\\_speed \(C++ member\), 246](#)  
[i2c\\_config\\_t::mode \(C++ member\), 246](#)  
[i2c\\_config\\_t::scl\\_io\\_num \(C++ member\), 246](#)  
[i2c\\_config\\_t::scl\\_pullup\\_en \(C++ member\), 246](#)  
[i2c\\_config\\_t::sda\\_io\\_num \(C++ member\), 246](#)  
[i2c\\_config\\_t::sda\\_pullup\\_en \(C++ member\), 246](#)  
[i2c\\_config\\_t::slave\\_addr \(C++ member\), 247](#)  
[I2C\\_DATA\\_MODE\\_LSB\\_FIRST \(C++ enumerator\), 247](#)  
[I2C\\_DATA\\_MODE\\_MAX \(C++ enumerator\), 247](#)  
[I2C\\_DATA\\_MODE\\_MSB\\_FIRST \(C++ enumerator\), 247](#)  
[i2c\\_driver\\_delete \(C++ function\), 237](#)  
[i2c\\_driver\\_install \(C++ function\), 237](#)  
[I2C\\_FIFO\\_LEN \(C macro\), 247](#)  
[i2c\\_filter\\_disable \(C++ function\), 243](#)  
[i2c\\_filter\\_enable \(C++ function\), 243](#)  
[i2c\\_get\\_data\\_mode \(C++ function\), 246](#)  
[i2c\\_get\\_data\\_timing \(C++ function\), 245](#)  
[i2c\\_get\\_period \(C++ function\), 243](#)  
[i2c\\_get\\_start\\_timing \(C++ function\), 244](#)  
[i2c\\_get\\_stop\\_timing \(C++ function\), 244](#)  
[i2c\\_get\\_timeout \(C++ function\), 245](#)  
[i2c\\_isr\\_free \(C++ function\), 238](#)  
[i2c\\_isr\\_register \(C++ function\), 238](#)  
[I2C\\_MASTER\\_ACK \(C++ enumerator\), 248](#)  
[I2C\\_MASTER\\_ACK\\_MAX \(C++ enumerator\), 248](#)  
[i2c\\_master\\_cmd\\_begin \(C++ function\), 241](#)  
[I2C\\_MASTER\\_LAST\\_NACK \(C++ enumerator\), 248](#)  
[I2C\\_MASTER\\_NACK \(C++ enumerator\), 248](#)  
[I2C\\_MASTER\\_READ \(C++ enumerator\), 247](#)  
[i2c\\_master\\_read \(C++ function\), 241](#)  
[i2c\\_master\\_read\\_byte \(C++ function\), 240](#)  
[i2c\\_master\\_start \(C++ function\), 239](#)  
[i2c\\_master\\_stop \(C++ function\), 241](#)  
[I2C\\_MASTER\\_WRITE \(C++ enumerator\), 247](#)  
[i2c\\_master\\_write \(C++ function\), 240](#)  
[i2c\\_master\\_write\\_byte \(C++ function\), 240](#)  
[I2C\\_MODE\\_MASTER \(C++ enumerator\), 247](#)  
[I2C\\_MODE\\_MAX \(C++ enumerator\), 247](#)  
[I2C\\_MODE\\_SLAVE \(C++ enumerator\), 247](#)  
[i2c\\_mode\\_t \(C++ type\), 247](#)  
[I2C\\_NUM\\_0 \(C++ enumerator\), 248](#)  
[I2C\\_NUM\\_1 \(C++ enumerator\), 248](#)  
[I2C\\_NUM\\_MAX \(C++ enumerator\), 248](#)  
[i2c\\_opmode\\_t \(C++ type\), 247](#)  
[i2c\\_param\\_config \(C++ function\), 238](#)  
[i2c\\_port\\_t \(C++ type\), 248](#)  
[i2c\\_reset\\_rx\\_fifo \(C++ function\), 238](#)  
[i2c\\_reset\\_tx\\_fifo \(C++ function\), 238](#)  
[i2c\\_rw\\_t \(C++ type\), 247](#)  
[i2c\\_set\\_data\\_mode \(C++ function\), 246](#)  
[i2c\\_set\\_data\\_timing \(C++ function\), 245](#)

- `i2c_set_period` (C++ function), 242
- `i2c_set_pin` (C++ function), 239
- `i2c_set_start_timing` (C++ function), 243
- `i2c_set_stop_timing` (C++ function), 244
- `i2c_set_timeout` (C++ function), 245
- `i2c_slave_read_buffer` (C++ function), 242
- `i2c_slave_write_buffer` (C++ function), 242
- `i2c_trans_mode_t` (C++ type), 247
- `i2s_adc_disable` (C++ function), 256
- `i2s_adc_enable` (C++ function), 255
- `I2S_BITS_PER_SAMPLE_16BIT` (C++ enumerator), 258
- `I2S_BITS_PER_SAMPLE_24BIT` (C++ enumerator), 258
- `I2S_BITS_PER_SAMPLE_32BIT` (C++ enumerator), 258
- `I2S_BITS_PER_SAMPLE_8BIT` (C++ enumerator), 258
- `i2s_bits_per_sample_t` (C++ type), 258
- `I2S_CHANNEL_FMT_ALL_LEFT` (C++ enumerator), 258
- `I2S_CHANNEL_FMT_ALL_RIGHT` (C++ enumerator), 258
- `I2S_CHANNEL_FMT_ONLY_LEFT` (C++ enumerator), 258
- `I2S_CHANNEL_FMT_ONLY_RIGHT` (C++ enumerator), 258
- `I2S_CHANNEL_FMT_RIGHT_LEFT` (C++ enumerator), 258
- `i2s_channel_fmt_t` (C++ type), 258
- `I2S_CHANNEL_MONO` (C++ enumerator), 258
- `I2S_CHANNEL_STEREO` (C++ enumerator), 258
- `i2s_channel_t` (C++ type), 258
- `I2S_COMM_FORMAT_I2S` (C++ enumerator), 258
- `I2S_COMM_FORMAT_I2S_LSB` (C++ enumerator), 258
- `I2S_COMM_FORMAT_I2S_MSB` (C++ enumerator), 258
- `I2S_COMM_FORMAT_PCM` (C++ enumerator), 258
- `I2S_COMM_FORMAT_PCM_LONG` (C++ enumerator), 258
- `I2S_COMM_FORMAT_PCM_SHORT` (C++ enumerator), 258
- `i2s_comm_format_t` (C++ type), 258
- `i2s_config_t` (C++ class), 256
- `i2s_config_t::bits_per_sample` (C++ member), 256
- `i2s_config_t::channel_format` (C++ member), 256
- `i2s_config_t::communication_format` (C++ member), 256
- `i2s_config_t::dma_buf_count` (C++ member), 257
- `i2s_config_t::dma_buf_len` (C++ member), 257
- `i2s_config_t::intr_alloc_flags` (C++ member), 256
- `i2s_config_t::mode` (C++ member), 256
- `i2s_config_t::sample_rate` (C++ member), 256
- `i2s_config_t::use_apll` (C++ member), 257
- `I2S_DAC_CHANNEL_BOTH_EN` (C++ enumerator), 260
- `I2S_DAC_CHANNEL_DISABLE` (C++ enumerator), 260
- `I2S_DAC_CHANNEL_LEFT_EN` (C++ enumerator), 260
- `I2S_DAC_CHANNEL_MAX` (C++ enumerator), 260
- `I2S_DAC_CHANNEL_RIGHT_EN` (C++ enumerator), 260
- `i2s_dac_mode_t` (C++ type), 260
- `i2s_driver_install` (C++ function), 251
- `i2s_driver_uninstall` (C++ function), 251
- `I2S_EVENT_DMA_ERROR` (C++ enumerator), 259
- `I2S_EVENT_MAX` (C++ enumerator), 260
- `I2S_EVENT_RX_DONE` (C++ enumerator), 259
- `i2s_event_t` (C++ class), 257
- `i2s_event_t::size` (C++ member), 257
- `i2s_event_t::type` (C++ member), 257
- `I2S_EVENT_TX_DONE` (C++ enumerator), 259
- `i2s_event_type_t` (C++ type), 259
- `i2s_isr_handle_t` (C++ type), 257
- `I2S_MODE_ADC_BUILT_IN` (C++ enumerator), 259
- `I2S_MODE_DAC_BUILT_IN` (C++ enumerator), 259
- `I2S_MODE_MASTER` (C++ enumerator), 259
- `I2S_MODE_PDM` (C++ enumerator), 259
- `I2S_MODE_RX` (C++ enumerator), 259
- `I2S_MODE_SLAVE` (C++ enumerator), 259
- `i2s_mode_t` (C++ type), 259
- `I2S_MODE_TX` (C++ enumerator), 259
- `I2S_NUM_0` (C++ enumerator), 259
- `I2S_NUM_1` (C++ enumerator), 259
- `I2S_NUM_MAX` (C++ enumerator), 259
- `i2s_pin_config_t` (C++ class), 257
- `i2s_pin_config_t::bck_io_num` (C++ member), 257
- `i2s_pin_config_t::data_in_num` (C++ member), 257
- `i2s_pin_config_t::data_out_num` (C++ member), 257
- `i2s_pin_config_t::ws_io_num` (C++ member), 257
- `I2S_PIN_NO_CHANGE` (C macro), 257
- `i2s_pop_sample` (C++ function), 253
- `i2s_port_t` (C++ type), 259
- `i2s_push_sample` (C++ function), 253
- `i2s_read` (C++ function), 253
- `i2s_read_bytes` (C++ function), 252

- `i2s_set_adc_mode` (C++ function), 255
  - `i2s_set_clk` (C++ function), 255
  - `i2s_set_dac_mode` (C++ function), 251
  - `i2s_set_pin` (C++ function), 250
  - `i2s_set_sample_rates` (C++ function), 254
  - `i2s_start` (C++ function), 254
  - `i2s_stop` (C++ function), 254
  - `i2s_write` (C++ function), 252
  - `i2s_write_bytes` (C++ function), 251
  - `i2s_write_expand` (C++ function), 252
  - `i2s_zero_dma_buffer` (C++ function), 255
  - `I_ADDI` (C macro), 817
  - `I_ADDR` (C macro), 816
  - `I_ANDI` (C macro), 817
  - `I_ANDR` (C macro), 816
  - `I_BGE` (C macro), 816
  - `I_BL` (C macro), 816
  - `I_BXFI` (C macro), 816
  - `I_BXFR` (C macro), 816
  - `I_BXI` (C macro), 816
  - `I_BXR` (C macro), 816
  - `I_BXZI` (C macro), 816
  - `I_BXZR` (C macro), 816
  - `I_DELAY` (C macro), 815
  - `I_END` (C macro), 815
  - `I_HALT` (C macro), 815
  - `I_LD` (C macro), 816
  - `I_LSHI` (C macro), 817
  - `I_LSHR` (C macro), 817
  - `I_MOVI` (C macro), 817
  - `I_MOVR` (C macro), 817
  - `I_ORI` (C macro), 817
  - `I_ORR` (C macro), 817
  - `I_RD_REG` (C macro), 816
  - `I_RSHI` (C macro), 817
  - `I_RSHR` (C macro), 817
  - `I_ST` (C macro), 815
  - `I_SUBI` (C macro), 817
  - `I_SUBR` (C macro), 816
  - `I_WR_REG` (C macro), 816
  - `intr_handle_data_t` (C++ type), 572
  - `intr_handle_t` (C++ type), 572
  - `intr_handler_t` (C++ type), 572
- L**
- `LEDC_APB_CLK` (C++ enumerator), 271
  - `LEDC_APB_CLK_HZ` (C macro), 270
  - `ledc_bind_channel_timer` (C++ function), 267
  - `LEDC_CHANNEL_0` (C++ enumerator), 271
  - `LEDC_CHANNEL_1` (C++ enumerator), 271
  - `LEDC_CHANNEL_2` (C++ enumerator), 271
  - `LEDC_CHANNEL_3` (C++ enumerator), 271
  - `LEDC_CHANNEL_4` (C++ enumerator), 271
  - `LEDC_CHANNEL_5` (C++ enumerator), 271
  - `LEDC_CHANNEL_6` (C++ enumerator), 271
  - `LEDC_CHANNEL_7` (C++ enumerator), 271
  - `ledc_channel_config` (C++ function), 264
  - `ledc_channel_config_t` (C++ class), 269
  - `ledc_channel_config_t::channel` (C++ member), 269
  - `ledc_channel_config_t::duty` (C++ member), 269
  - `ledc_channel_config_t::gpio_num` (C++ member), 269
  - `ledc_channel_config_t::intr_type` (C++ member), 269
  - `ledc_channel_config_t::speed_mode` (C++ member), 269
  - `ledc_channel_config_t::timer_sel` (C++ member), 269
  - `LEDC_CHANNEL_MAX` (C++ enumerator), 271
  - `ledc_channel_t` (C++ type), 271
  - `ledc_clk_src_t` (C++ type), 271
  - `LEDC_DUTY_DIR_DECREASE` (C++ enumerator), 271
  - `LEDC_DUTY_DIR_INCREASE` (C++ enumerator), 271
  - `ledc_duty_direction_t` (C++ type), 271
  - `LEDC_ERR_DUTY` (C macro), 270
  - `ledc_fade_func_install` (C++ function), 269
  - `ledc_fade_func_uninstall` (C++ function), 269
  - `LEDC_FADE_MAX` (C++ enumerator), 272
  - `ledc_fade_mode_t` (C++ type), 272
  - `LEDC_FADE_NO_WAIT` (C++ enumerator), 272
  - `ledc_fade_start` (C++ function), 269
  - `LEDC_FADE_WAIT_DONE` (C++ enumerator), 272
  - `ledc_get_duty` (C++ function), 265
  - `ledc_get_freq` (C++ function), 265
  - `LEDC_HIGH_SPEED_MODE` (C++ enumerator), 270
  - `LEDC_INTR_DISABLE` (C++ enumerator), 270
  - `LEDC_INTR_FADE_END` (C++ enumerator), 270
  - `ledc_intr_type_t` (C++ type), 270
  - `ledc_isr_handle_t` (C++ type), 270
  - `ledc_isr_register` (C++ function), 266
  - `LEDC_LOW_SPEED_MODE` (C++ enumerator), 270
  - `ledc_mode_t` (C++ type), 270
  - `LEDC_REF_CLK_HZ` (C macro), 270
  - `LEDC_REF_TICK` (C++ enumerator), 271
  - `ledc_set_duty` (C++ function), 265
  - `ledc_set_fade` (C++ function), 266
  - `ledc_set_fade_with_step` (C++ function), 268
  - `ledc_set_fade_with_time` (C++ function), 268
  - `ledc_set_freq` (C++ function), 265
  - `LEDC_SPEED_MODE_MAX` (C++ enumerator), 270
  - `ledc_stop` (C++ function), 264
  - `LEDC_TIMER_0` (C++ enumerator), 271
  - `LEDC_TIMER_1` (C++ enumerator), 271
  - `LEDC_TIMER_10_BIT` (C++ enumerator), 272
  - `LEDC_TIMER_11_BIT` (C++ enumerator), 272
  - `LEDC_TIMER_12_BIT` (C++ enumerator), 272

- LEDC\_TIMER\_13\_BIT (C++ enumerator), 272  
 LEDC\_TIMER\_14\_BIT (C++ enumerator), 272  
 LEDC\_TIMER\_15\_BIT (C++ enumerator), 272  
 LEDC\_TIMER\_2 (C++ enumerator), 271  
 LEDC\_TIMER\_3 (C++ enumerator), 271  
 ledc\_timer\_bit\_t (C++ type), 271  
 ledc\_timer\_config (C++ function), 264  
 ledc\_timer\_config\_t (C++ class), 270  
 ledc\_timer\_config\_t::bit\_num (C++ member), 270  
 ledc\_timer\_config\_t::duty\_resolution (C++ member), 270  
 ledc\_timer\_config\_t::freq\_hz (C++ member), 270  
 ledc\_timer\_config\_t::speed\_mode (C++ member), 270  
 ledc\_timer\_config\_t::timer\_num (C++ member), 270  
 ledc\_timer\_pause (C++ function), 267  
 ledc\_timer\_resume (C++ function), 267  
 ledc\_timer\_rst (C++ function), 267  
 ledc\_timer\_set (C++ function), 266  
 ledc\_timer\_t (C++ type), 271  
 ledc\_update\_duty (C++ function), 264  
 LOG\_COLOR\_D (C macro), 586  
 LOG\_COLOR\_E (C macro), 586  
 LOG\_COLOR\_I (C macro), 586  
 LOG\_COLOR\_V (C macro), 586  
 LOG\_COLOR\_W (C macro), 586  
 LOG\_FORMAT (C macro), 586  
 LOG\_LOCAL\_LEVEL (C macro), 586  
 LOG\_RESET\_COLOR (C macro), 586
- ## M
- M\_BGE (C macro), 817  
 M\_BL (C macro), 817  
 M\_BX (C macro), 817  
 M\_BXF (C macro), 818  
 M\_BXZ (C macro), 817  
 M\_LABEL (C macro), 817  
 MALLOC\_CAP\_32BIT (C macro), 552  
 MALLOC\_CAP\_8BIT (C macro), 552  
 MALLOC\_CAP\_DEFAULT (C macro), 552  
 MALLOC\_CAP\_DMA (C macro), 552  
 MALLOC\_CAP\_EXEC (C macro), 552  
 MALLOC\_CAP\_INTERNAL (C macro), 552  
 MALLOC\_CAP\_INVALID (C macro), 552  
 MALLOC\_CAP\_PID2 (C macro), 552  
 MALLOC\_CAP\_PID3 (C macro), 552  
 MALLOC\_CAP\_PID4 (C macro), 552  
 MALLOC\_CAP\_PID5 (C macro), 552  
 MALLOC\_CAP\_PID6 (C macro), 552  
 MALLOC\_CAP\_PID7 (C macro), 552  
 MALLOC\_CAP\_SPIRAM (C macro), 552  
 MCPWM0A (C++ enumerator), 285  
 MCPWM0B (C++ enumerator), 285  
 MCPWM1A (C++ enumerator), 285  
 MCPWM1B (C++ enumerator), 285  
 MCPWM2A (C++ enumerator), 285  
 MCPWM2B (C++ enumerator), 285  
 mcpwm\_action\_on\_pwmxa\_t (C++ type), 288  
 mcpwm\_action\_on\_pwmxb\_t (C++ type), 288  
 MCPWM\_ACTIVE\_HIGH\_COMPLIMENT\_MODE (C++ enumerator), 289  
 MCPWM\_ACTIVE\_HIGH\_MODE (C++ enumerator), 289  
 MCPWM\_ACTIVE\_LOW\_COMPLIMENT\_MODE (C++ enumerator), 289  
 MCPWM\_ACTIVE\_LOW\_MODE (C++ enumerator), 289  
 MCPWM\_ACTIVE\_RED\_FED\_FROM\_PWMXA (C++ enumerator), 289  
 MCPWM\_ACTIVE\_RED\_FED\_FROM\_PWMXB (C++ enumerator), 289  
 MCPWM\_BYPASS\_FED (C++ enumerator), 289  
 MCPWM\_BYPASS\_RED (C++ enumerator), 289  
 MCPWM\_CAP\_0 (C++ enumerator), 286  
 MCPWM\_CAP\_1 (C++ enumerator), 286  
 MCPWM\_CAP\_2 (C++ enumerator), 286  
 mcpwm\_capture\_disable (C++ function), 282  
 mcpwm\_capture\_enable (C++ function), 281  
 mcpwm\_capture\_on\_edge\_t (C++ type), 289  
 mcpwm\_capture\_signal\_get\_edge (C++ function), 282  
 mcpwm\_capture\_signal\_get\_value (C++ function), 282  
 mcpwm\_capture\_signal\_t (C++ type), 288  
 mcpwm\_carrier\_config\_t (C++ class), 284  
 mcpwm\_carrier\_config\_t::carrier\_duty (C++ member), 285  
 mcpwm\_carrier\_config\_t::carrier\_ivt\_mode (C++ member), 285  
 mcpwm\_carrier\_config\_t::carrier\_os\_mode (C++ member), 285  
 mcpwm\_carrier\_config\_t::carrier\_period (C++ member), 285  
 mcpwm\_carrier\_config\_t::pulse\_width\_in\_os (C++ member), 285  
 mcpwm\_carrier\_disable (C++ function), 278  
 mcpwm\_carrier\_enable (C++ function), 277  
 mcpwm\_carrier\_init (C++ function), 277  
 mcpwm\_carrier\_oneshot\_mode\_disable (C++ function), 279  
 mcpwm\_carrier\_oneshot\_mode\_enable (C++ function), 279  
 mcpwm\_carrier\_os\_t (C++ type), 287  
 MCPWM\_CARRIER\_OUT\_IVT\_DIS (C++ enumerator), 287  
 MCPWM\_CARRIER\_OUT\_IVT\_EN (C++ enumerator), 287

- mcpwm\_carrier\_out\_ivt\_t (C++ type), 287
- mcpwm\_carrier\_output\_invert (C++ function), 279
- mcpwm\_carrier\_set\_duty\_cycle (C++ function), 278
- mcpwm\_carrier\_set\_period (C++ function), 278
- mcpwm\_config\_t (C++ class), 284
- mcpwm\_config\_t::cmpr\_a (C++ member), 284
- mcpwm\_config\_t::cmpr\_b (C++ member), 284
- mcpwm\_config\_t::counter\_mode (C++ member), 284
- mcpwm\_config\_t::duty\_mode (C++ member), 284
- mcpwm\_config\_t::frequency (C++ member), 284
- MCPWM\_COUNTER\_MAX (C++ enumerator), 287
- mcpwm\_counter\_type\_t (C++ type), 286
- mcpwm\_deadtime\_disable (C++ function), 280
- mcpwm\_deadtime\_enable (C++ function), 279
- MCPWM\_DEADTIME\_TYPE\_MAX (C++ enumerator), 289
- mcpwm\_deadtime\_type\_t (C++ type), 289
- MCPWM\_DOWN\_COUNTER (C++ enumerator), 286
- MCPWM\_DUTY\_MODE\_0 (C++ enumerator), 287
- MCPWM\_DUTY\_MODE\_1 (C++ enumerator), 287
- MCPWM\_DUTY\_MODE\_MAX (C++ enumerator), 287
- mcpwm\_duty\_type\_t (C++ type), 287
- MCPWM\_FAULT\_0 (C++ enumerator), 285
- MCPWM\_FAULT\_1 (C++ enumerator), 285
- MCPWM\_FAULT\_2 (C++ enumerator), 285
- mcpwm\_fault\_deinit (C++ function), 281
- mcpwm\_fault\_init (C++ function), 280
- mcpwm\_fault\_input\_level\_t (C++ type), 288
- mcpwm\_fault\_set\_cyc\_mode (C++ function), 281
- mcpwm\_fault\_set\_oneshot\_mode (C++ function), 280
- mcpwm\_fault\_signal\_t (C++ type), 287
- MCPWM\_FORCE\_MCPWMXA\_HIGH (C++ enumerator), 288
- MCPWM\_FORCE\_MCPWMXA\_LOW (C++ enumerator), 288
- MCPWM\_FORCE\_MCPWMXB\_HIGH (C++ enumerator), 288
- MCPWM\_FORCE\_MCPWMXB\_LOW (C++ enumerator), 288
- mcpwm\_get\_duty (C++ function), 276
- mcpwm\_get\_frequency (C++ function), 276
- mcpwm\_gpio\_init (C++ function), 274
- MCPWM\_HIGH\_LEVEL\_TGR (C++ enumerator), 288
- mcpwm\_init (C++ function), 274
- mcpwm\_io\_signals\_t (C++ type), 285
- mcpwm\_isr\_register (C++ function), 283
- MCPWM\_LOW\_LEVEL\_TGR (C++ enumerator), 288
- MCPWM\_NEG\_EDGE (C++ enumerator), 289
- MCPWM\_NO\_CHANGE\_IN\_MCPWMXA (C++ enumerator), 288
- MCPWM\_NO\_CHANGE\_IN\_MCPWMXB (C++ enumerator), 288
- MCPWM\_ONESHOT\_MODE\_DIS (C++ enumerator), 287
- MCPWM\_ONESHOT\_MODE\_EN (C++ enumerator), 287
- mcpwm\_operator\_t (C++ type), 286
- MCPWM\_OPR\_A (C++ enumerator), 286
- MCPWM\_OPR\_B (C++ enumerator), 286
- MCPWM\_OPR\_MAX (C++ enumerator), 286
- mcpwm\_pin\_config\_t (C++ class), 283
- mcpwm\_pin\_config\_t::mcpwm0a\_out\_num (C++ member), 283
- mcpwm\_pin\_config\_t::mcpwm0b\_out\_num (C++ member), 283
- mcpwm\_pin\_config\_t::mcpwm1a\_out\_num (C++ member), 283
- mcpwm\_pin\_config\_t::mcpwm1b\_out\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm2a\_out\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm2b\_out\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_cap0\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_cap1\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_cap2\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_fault0\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_fault1\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_fault2\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_sync0\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_sync1\_in\_num (C++ member), 284
- mcpwm\_pin\_config\_t::mcpwm\_sync2\_in\_num (C++ member), 284
- MCPWM\_POS\_EDGE (C++ enumerator), 289
- MCPWM\_SELECT\_CAP0 (C++ enumerator), 288
- MCPWM\_SELECT\_CAP1 (C++ enumerator), 288
- MCPWM\_SELECT\_CAP2 (C++ enumerator), 288
- MCPWM\_SELECT\_F0 (C++ enumerator), 287
- MCPWM\_SELECT\_F1 (C++ enumerator), 288
- MCPWM\_SELECT\_F2 (C++ enumerator), 288
- MCPWM\_SELECT\_SYNC0 (C++ enumerator), 287
- MCPWM\_SELECT\_SYNC1 (C++ enumerator), 287
- MCPWM\_SELECT\_SYNC2 (C++ enumerator), 287
- mcpwm\_set\_duty (C++ function), 275
- mcpwm\_set\_duty\_in\_us (C++ function), 275
- mcpwm\_set\_duty\_type (C++ function), 275

- mcpwm\_set\_frequency (C++ function), 274
- mcpwm\_set\_pin (C++ function), 274
- mcpwm\_set\_signal\_high (C++ function), 276
- mcpwm\_set\_signal\_low (C++ function), 276
- mcpwm\_start (C++ function), 277
- mcpwm\_stop (C++ function), 277
- MCPWM\_SYNC\_0 (C++ enumerator), 285
- MCPWM\_SYNC\_1 (C++ enumerator), 285
- MCPWM\_SYNC\_2 (C++ enumerator), 285
- mcpwm\_sync\_disable (C++ function), 283
- mcpwm\_sync\_enable (C++ function), 282
- mcpwm\_sync\_signal\_t (C++ type), 287
- MCPWM\_TIMER\_0 (C++ enumerator), 286
- MCPWM\_TIMER\_1 (C++ enumerator), 286
- MCPWM\_TIMER\_2 (C++ enumerator), 286
- MCPWM\_TIMER\_MAX (C++ enumerator), 286
- mcpwm\_timer\_t (C++ type), 286
- MCPWM\_TOG\_MCPWMXA (C++ enumerator), 288
- MCPWM\_TOG\_MCPWMB (C++ enumerator), 288
- MCPWM\_UNIT\_0 (C++ enumerator), 286
- MCPWM\_UNIT\_1 (C++ enumerator), 286
- MCPWM\_UNIT\_MAX (C++ enumerator), 286
- mcpwm\_unit\_t (C++ type), 286
- MCPWM\_UP\_COUNTER (C++ enumerator), 286
- MCPWM\_UP\_DOWN\_COUNTER (C++ enumerator), 287
- mdns\_free (C++ function), 395
- mdns\_handle\_system\_event (C++ function), 400
- mdns\_hostname\_set (C++ function), 395
- mdns\_init (C++ function), 395
- mdns\_instance\_name\_set (C++ function), 395
- mdns\_ip\_addr\_s (C++ class), 401
- mdns\_ip\_addr\_s::addr (C++ member), 401
- mdns\_ip\_addr\_s::next (C++ member), 401
- mdns\_ip\_addr\_t (C++ type), 402
- MDNS\_IP\_PROTOCOL\_MAX (C++ enumerator), 402
- mdns\_ip\_protocol\_t (C++ type), 402
- MDNS\_IP\_PROTOCOL\_V4 (C++ enumerator), 402
- MDNS\_IP\_PROTOCOL\_V6 (C++ enumerator), 402
- mdns\_query (C++ function), 398
- mdns\_query\_a (C++ function), 400
- mdns\_query\_aaaa (C++ function), 400
- mdns\_query\_ptr (C++ function), 398
- mdns\_query\_results\_free (C++ function), 398
- mdns\_query\_srv (C++ function), 399
- mdns\_query\_txt (C++ function), 399
- mdns\_result\_s (C++ class), 401
- mdns\_result\_s::addr (C++ member), 401
- mdns\_result\_s::hostname (C++ member), 401
- mdns\_result\_s::instance\_name (C++ member), 401
- mdns\_result\_s::ip\_protocol (C++ member), 401
- mdns\_result\_s::next (C++ member), 401
- mdns\_result\_s::port (C++ member), 401
- mdns\_result\_s::tcpip\_if (C++ member), 401
- mdns\_result\_s::txt (C++ member), 401
- mdns\_result\_s::txt\_count (C++ member), 401
- mdns\_result\_t (C++ type), 402
- mdns\_service\_add (C++ function), 395
- mdns\_service\_instance\_name\_set (C++ function), 396
- mdns\_service\_port\_set (C++ function), 396
- mdns\_service\_remove (C++ function), 396
- mdns\_service\_remove\_all (C++ function), 398
- mdns\_service\_txt\_item\_remove (C++ function), 397
- mdns\_service\_txt\_item\_set (C++ function), 397
- mdns\_service\_txt\_set (C++ function), 397
- mdns\_txt\_item\_t (C++ class), 401
- mdns\_txt\_item\_t::key (C++ member), 401
- mdns\_txt\_item\_t::value (C++ member), 401
- MDNS\_TYPE\_A (C macro), 402
- MDNS\_TYPE\_AAAA (C macro), 402
- MDNS\_TYPE\_ANY (C macro), 402
- MDNS\_TYPE\_NSEC (C macro), 402
- MDNS\_TYPE\_OPT (C macro), 402
- MDNS\_TYPE\_PTR (C macro), 402
- MDNS\_TYPE\_SRV (C macro), 402
- MDNS\_TYPE\_TXT (C macro), 402
- multi\_heap\_check (C++ function), 556
- multi\_heap\_dump (C++ function), 556
- multi\_heap\_free (C++ function), 554
- multi\_heap\_free\_size (C++ function), 556
- multi\_heap\_get\_allocated\_size (C++ function), 555
- multi\_heap\_get\_info (C++ function), 556
- multi\_heap\_handle\_t (C++ type), 557
- multi\_heap\_info\_t (C++ class), 557
- multi\_heap\_info\_t::allocated\_blocks (C++ member), 557
- multi\_heap\_info\_t::free\_blocks (C++ member), 557
- multi\_heap\_info\_t::largest\_free\_block (C++ member), 557
- multi\_heap\_info\_t::minimum\_free\_bytes (C++ member), 557
- multi\_heap\_info\_t::total\_allocated\_bytes (C++ member), 557
- multi\_heap\_info\_t::total\_blocks (C++ member), 557
- multi\_heap\_info\_t::total\_free\_bytes (C++ member), 557
- multi\_heap\_malloc (C++ function), 554
- multi\_heap\_minimum\_free\_size (C++ function), 556
- multi\_heap\_realloc (C++ function), 555
- multi\_heap\_register (C++ function), 555

multi\_heap\_set\_lock (C++ function), 555

## N

nvs\_close (C++ function), 429

nvs\_commit (C++ function), 429

NVS\_DEFAULT\_PART\_NAME (C macro), 430

nvs\_erase\_all (C++ function), 429

nvs\_erase\_key (C++ function), 428

nvs\_flash\_deinit (C++ function), 423

nvs\_flash\_deinit\_partition (C++ function), 423

nvs\_flash\_erase (C++ function), 424

nvs\_flash\_erase\_partition (C++ function), 424

nvs\_flash\_init (C++ function), 423

nvs\_flash\_init\_partition (C++ function), 423

nvs\_get\_blob (C++ function), 427

nvs\_get\_i16 (C++ function), 426

nvs\_get\_i32 (C++ function), 426

nvs\_get\_i64 (C++ function), 426

nvs\_get\_i8 (C++ function), 425

nvs\_get\_str (C++ function), 426

nvs\_get\_u16 (C++ function), 426

nvs\_get\_u32 (C++ function), 426

nvs\_get\_u64 (C++ function), 426

nvs\_get\_u8 (C++ function), 426

nvs\_handle (C++ type), 430

nvs\_open (C++ function), 427

nvs\_open\_from\_partition (C++ function), 427

nvs\_open\_mode (C++ type), 431

NVS\_READONLY (C++ enumerator), 431

NVS\_READWRITE (C++ enumerator), 431

nvs\_set\_blob (C++ function), 428

nvs\_set\_i16 (C++ function), 425

nvs\_set\_i32 (C++ function), 425

nvs\_set\_i64 (C++ function), 425

nvs\_set\_i8 (C++ function), 424

nvs\_set\_str (C++ function), 425

nvs\_set\_u16 (C++ function), 425

nvs\_set\_u32 (C++ function), 425

nvs\_set\_u64 (C++ function), 425

nvs\_set\_u8 (C++ function), 425

## O

OTA\_SIZE\_UNKNOWN (C macro), 610

## P

PCNT\_CHANNEL\_0 (C++ enumerator), 298

PCNT\_CHANNEL\_1 (C++ enumerator), 298

PCNT\_CHANNEL\_MAX (C++ enumerator), 298

pcnt\_channel\_t (C++ type), 298

pcnt\_config\_t (C++ class), 296

pcnt\_config\_t::channel (C++ member), 296

pcnt\_config\_t::counter\_h\_lim (C++ member), 296

pcnt\_config\_t::counter\_l\_lim (C++ member), 296

pcnt\_config\_t::ctrl\_gpio\_num (C++ member), 296

pcnt\_config\_t::hctrl\_mode (C++ member), 296

pcnt\_config\_t::lctrl\_mode (C++ member), 296

pcnt\_config\_t::neg\_mode (C++ member), 296

pcnt\_config\_t::pos\_mode (C++ member), 296

pcnt\_config\_t::pulse\_gpio\_num (C++ member), 296

pcnt\_config\_t::unit (C++ member), 296

PCNT\_COUNT\_DEC (C++ enumerator), 297

PCNT\_COUNT\_DIS (C++ enumerator), 297

PCNT\_COUNT\_INC (C++ enumerator), 297

PCNT\_COUNT\_MAX (C++ enumerator), 297

pcnt\_count\_mode\_t (C++ type), 297

pcnt\_counter\_clear (C++ function), 292

pcnt\_counter\_pause (C++ function), 292

pcnt\_counter\_resume (C++ function), 292

pcnt\_ctrl\_mode\_t (C++ type), 297

pcnt\_event\_disable (C++ function), 293

pcnt\_event\_enable (C++ function), 293

PCNT\_EVT\_H\_LIM (C++ enumerator), 298

PCNT\_EVT\_L\_LIM (C++ enumerator), 298

PCNT\_EVT\_MAX (C++ enumerator), 298

PCNT\_EVT\_THRES\_0 (C++ enumerator), 298

PCNT\_EVT\_THRES\_1 (C++ enumerator), 298

pcnt\_evt\_type\_t (C++ type), 298

PCNT\_EVT\_ZERO (C++ enumerator), 298

pcnt\_filter\_disable (C++ function), 295

pcnt\_filter\_enable (C++ function), 294

pcnt\_get\_counter\_value (C++ function), 291

pcnt\_get\_event\_value (C++ function), 294

pcnt\_get\_filter\_value (C++ function), 295

pcnt\_intr\_disable (C++ function), 293

pcnt\_intr\_enable (C++ function), 292

pcnt\_isr\_handle\_t (C++ type), 297

pcnt\_isr\_register (C++ function), 294

PCNT\_MODE\_DISABLE (C++ enumerator), 297

PCNT\_MODE\_KEEP (C++ enumerator), 297

PCNT\_MODE\_MAX (C++ enumerator), 297

PCNT\_MODE\_REVERSE (C++ enumerator), 297

PCNT\_PIN\_NOT\_USED (C macro), 296

pcnt\_set\_event\_value (C++ function), 293

pcnt\_set\_filter\_value (C++ function), 295

pcnt\_set\_mode (C++ function), 295

pcnt\_set\_pin (C++ function), 294

PCNT\_UNIT\_0 (C++ enumerator), 297

PCNT\_UNIT\_1 (C++ enumerator), 297

PCNT\_UNIT\_2 (C++ enumerator), 297

- PCNT\_UNIT\_3 (C++ *enumerator*), 297
- PCNT\_UNIT\_4 (C++ *enumerator*), 297
- PCNT\_UNIT\_5 (C++ *enumerator*), 298
- PCNT\_UNIT\_6 (C++ *enumerator*), 298
- PCNT\_UNIT\_7 (C++ *enumerator*), 298
- pcnt\_unit\_config (C++ *function*), 291
- PCNT\_UNIT\_MAX (C++ *enumerator*), 298
- pcnt\_unit\_t (C++ *type*), 297
- pcQueueGetName (C++ *function*), 481
- pcTaskGetTaskName (C++ *function*), 461
- pcTimerGetTimerName (C++ *function*), 519
- PDM\_PCM\_CONV\_DISABLE (C++ *enumerator*), 259
- PDM\_PCM\_CONV\_ENABLE (C++ *enumerator*), 259
- pdm\_pcm\_conv\_t (C++ *type*), 259
- PDM\_SAMPLE\_RATE\_RATIO\_128 (C++ *enumerator*), 259
- PDM\_SAMPLE\_RATE\_RATIO\_64 (C++ *enumerator*), 259
- pdm\_sample\_rate\_ratio\_t (C++ *type*), 259
- PendedFunction\_t (C++ *type*), 529
- PHY0 (C++ *enumerator*), 198
- PHY1 (C++ *enumerator*), 198
- PHY10 (C++ *enumerator*), 198
- PHY11 (C++ *enumerator*), 198
- PHY12 (C++ *enumerator*), 198
- PHY13 (C++ *enumerator*), 198
- PHY14 (C++ *enumerator*), 198
- PHY15 (C++ *enumerator*), 198
- PHY16 (C++ *enumerator*), 198
- PHY17 (C++ *enumerator*), 199
- PHY18 (C++ *enumerator*), 199
- PHY19 (C++ *enumerator*), 199
- PHY2 (C++ *enumerator*), 198
- PHY20 (C++ *enumerator*), 199
- PHY21 (C++ *enumerator*), 199
- PHY22 (C++ *enumerator*), 199
- PHY23 (C++ *enumerator*), 199
- PHY24 (C++ *enumerator*), 199
- PHY25 (C++ *enumerator*), 199
- PHY26 (C++ *enumerator*), 199
- PHY27 (C++ *enumerator*), 199
- PHY28 (C++ *enumerator*), 199
- PHY29 (C++ *enumerator*), 199
- PHY3 (C++ *enumerator*), 198
- PHY30 (C++ *enumerator*), 199
- PHY31 (C++ *enumerator*), 199
- PHY4 (C++ *enumerator*), 198
- PHY5 (C++ *enumerator*), 198
- PHY6 (C++ *enumerator*), 198
- PHY7 (C++ *enumerator*), 198
- PHY8 (C++ *enumerator*), 198
- PHY9 (C++ *enumerator*), 198
- phy\_lan8720\_check\_phy\_init (C++ *function*), 200
- phy\_lan8720\_default\_ethernet\_config (C++ *member*), 193
- phy\_lan8720\_dump\_registers (C++ *function*), 200
- phy\_lan8720\_get\_duplex\_mode (C++ *function*), 200
- phy\_lan8720\_get\_speed\_mode (C++ *function*), 200
- phy\_lan8720\_init (C++ *function*), 201
- phy\_lan8720\_power\_enable (C++ *function*), 201
- phy\_mii\_check\_link\_status (C++ *function*), 199
- phy\_mii\_enable\_flow\_ctrl (C++ *function*), 199
- phy\_mii\_get\_partner\_pause\_enable (C++ *function*), 199
- phy\_rmii\_configure\_data\_interface\_pins (C++ *function*), 199
- phy\_rmii\_smi\_configure\_pins (C++ *function*), 199
- phy\_tlk110\_check\_phy\_init (C++ *function*), 200
- phy\_tlk110\_default\_ethernet\_config (C++ *member*), 193
- phy\_tlk110\_dump\_registers (C++ *function*), 200
- phy\_tlk110\_get\_duplex\_mode (C++ *function*), 200
- phy\_tlk110\_get\_speed\_mode (C++ *function*), 200
- phy\_tlk110\_init (C++ *function*), 200
- phy\_tlk110\_power\_enable (C++ *function*), 200
- pvTaskGetThreadLocalStoragePointer (C++ *function*), 462
- pvTimerGetTimerID (C++ *function*), 515
- pxTaskGetStackStart (C++ *function*), 461
- ## Q
- QueueHandle\_t (C++ *type*), 497
- QueueSetHandle\_t (C++ *type*), 497
- QueueSetMemberHandle\_t (C++ *type*), 497
- ## R
- R0 (C *macro*), 815
- R1 (C *macro*), 815
- R2 (C *macro*), 815
- R3 (C *macro*), 815
- RINGBUF\_TYPE\_ALLOWSPLIT (C++ *enumerator*), 543
- RINGBUF\_TYPE\_BYTEBUF (C++ *enumerator*), 544
- RINGBUF\_TYPE\_NOSPLIT (C++ *enumerator*), 543
- ringbuf\_type\_t (C++ *type*), 543
- RingbufHandle\_t (C++ *type*), 543
- RMT\_BASECLK\_APB (C++ *enumerator*), 316
- RMT\_BASECLK\_MAX (C++ *enumerator*), 316

RMT\_BASECLK\_REF (C++ enumerator), 316  
RMT\_CARRIER\_LEVEL\_HIGH (C++ enumerator), 317  
RMT\_CARRIER\_LEVEL\_LOW (C++ enumerator), 317  
RMT\_CARRIER\_LEVEL\_MAX (C++ enumerator), 317  
rmt\_carrier\_level\_t (C++ type), 316  
RMT\_CHANNEL\_0 (C++ enumerator), 315  
RMT\_CHANNEL\_1 (C++ enumerator), 315  
RMT\_CHANNEL\_2 (C++ enumerator), 315  
RMT\_CHANNEL\_3 (C++ enumerator), 315  
RMT\_CHANNEL\_4 (C++ enumerator), 315  
RMT\_CHANNEL\_5 (C++ enumerator), 316  
RMT\_CHANNEL\_6 (C++ enumerator), 316  
RMT\_CHANNEL\_7 (C++ enumerator), 316  
RMT\_CHANNEL\_MAX (C++ enumerator), 316  
rmt\_channel\_t (C++ type), 315  
rmt\_clr\_intr\_enable\_mask (C++ function), 310  
rmt\_config (C++ function), 311  
rmt\_config\_t (C++ class), 315  
rmt\_config\_t::channel (C++ member), 315  
rmt\_config\_t::clk\_div (C++ member), 315  
rmt\_config\_t::gpio\_num (C++ member), 315  
rmt\_config\_t::mem\_block\_num (C++ member), 315  
rmt\_config\_t::rmt\_mode (C++ member), 315  
rmt\_config\_t::rx\_config (C++ member), 315  
rmt\_config\_t::tx\_config (C++ member), 315  
RMT\_DATA\_MODE\_FIFO (C++ enumerator), 316  
RMT\_DATA\_MODE\_MAX (C++ enumerator), 316  
RMT\_DATA\_MODE\_MEM (C++ enumerator), 316  
rmt\_data\_mode\_t (C++ type), 316  
rmt\_driver\_install (C++ function), 312  
rmt\_driver\_uninstall (C++ function), 313  
rmt\_fill\_tx\_items (C++ function), 312  
rmt\_get\_clk\_div (C++ function), 304  
rmt\_get\_mem\_block\_num (C++ function), 305  
rmt\_get\_mem\_pd (C++ function), 306  
rmt\_get\_memory\_owner (C++ function), 307  
rmt\_get\_ringbuf\_handle (C++ function), 314  
rmt\_get\_rx\_idle\_thresh (C++ function), 304  
rmt\_get\_source\_clk (C++ function), 309  
rmt\_get\_status (C++ function), 309  
rmt\_get\_tx\_loop\_mode (C++ function), 308  
RMT\_IDLE\_LEVEL\_HIGH (C++ enumerator), 316  
RMT\_IDLE\_LEVEL\_LOW (C++ enumerator), 316  
RMT\_IDLE\_LEVEL\_MAX (C++ enumerator), 316  
rmt\_idle\_level\_t (C++ type), 316  
rmt\_isr\_deregister (C++ function), 312  
rmt\_isr\_handle\_t (C++ type), 315  
rmt\_isr\_register (C++ function), 311  
RMT\_MEM\_BLOCK\_BYTE\_NUM (C macro), 315  
RMT\_MEM\_ITEM\_NUM (C macro), 315  
RMT\_MEM\_OWNER\_MAX (C++ enumerator), 316  
RMT\_MEM\_OWNER\_RX (C++ enumerator), 316  
rmt\_mem\_owner\_t (C++ type), 316  
RMT\_MEM\_OWNER\_TX (C++ enumerator), 316  
rmt\_memory\_rw\_rst (C++ function), 307  
RMT\_MODE\_MAX (C++ enumerator), 316  
RMT\_MODE\_RX (C++ enumerator), 316  
rmt\_mode\_t (C++ type), 316  
RMT\_MODE\_TX (C++ enumerator), 316  
rmt\_rx\_config\_t (C++ class), 314  
rmt\_rx\_config\_t::filter\_en (C++ member), 314  
rmt\_rx\_config\_t::filter\_ticks\_thresh (C++ member), 314  
rmt\_rx\_config\_t::idle\_threshold (C++ member), 314  
rmt\_rx\_start (C++ function), 307  
rmt\_rx\_stop (C++ function), 307  
rmt\_set\_clk\_div (C++ function), 304  
rmt\_set\_err\_intr\_en (C++ function), 310  
rmt\_set\_idle\_level (C++ function), 309  
rmt\_set\_intr\_enable\_mask (C++ function), 310  
rmt\_set\_mem\_block\_num (C++ function), 305  
rmt\_set\_mem\_pd (C++ function), 306  
rmt\_set\_memory\_owner (C++ function), 307  
rmt\_set\_pin (C++ function), 311  
rmt\_set\_rx\_filter (C++ function), 308  
rmt\_set\_rx\_idle\_thresh (C++ function), 304  
rmt\_set\_rx\_intr\_en (C++ function), 310  
rmt\_set\_source\_clk (C++ function), 309  
rmt\_set\_tx\_carrier (C++ function), 305  
rmt\_set\_tx\_intr\_en (C++ function), 310  
rmt\_set\_tx\_loop\_mode (C++ function), 308  
rmt\_set\_tx\_thr\_intr\_en (C++ function), 311  
rmt\_source\_clk\_t (C++ type), 316  
rmt\_tx\_config\_t (C++ class), 314  
rmt\_tx\_config\_t::carrier\_duty\_percent (C++ member), 314  
rmt\_tx\_config\_t::carrier\_en (C++ member), 314  
rmt\_tx\_config\_t::carrier\_freq\_hz (C++ member), 314  
rmt\_tx\_config\_t::carrier\_level (C++ member), 314  
rmt\_tx\_config\_t::idle\_level (C++ member), 314  
rmt\_tx\_config\_t::idle\_output\_en (C++ member), 314  
rmt\_tx\_config\_t::loop\_en (C++ member), 314  
rmt\_tx\_start (C++ function), 306  
rmt\_tx\_stop (C++ function), 306  
rmt\_wait\_tx\_done (C++ function), 313  
rmt\_write\_items (C++ function), 313  
rtc\_gpio\_deinit (C++ function), 228  
rtc\_gpio\_desc\_t (C++ class), 231  
rtc\_gpio\_desc\_t::drv\_s (C++ member), 232  
rtc\_gpio\_desc\_t::drv\_v (C++ member), 232

- `rtc_gpio_desc_t::func` (C++ member), 231
  - `rtc_gpio_desc_t::hold` (C++ member), 232
  - `rtc_gpio_desc_t::hold_force` (C++ member), 232
  - `rtc_gpio_desc_t::ie` (C++ member), 231
  - `rtc_gpio_desc_t::mux` (C++ member), 231
  - `rtc_gpio_desc_t::pulldown` (C++ member), 231
  - `rtc_gpio_desc_t::pullup` (C++ member), 231
  - `rtc_gpio_desc_t::reg` (C++ member), 231
  - `rtc_gpio_desc_t::rtc_num` (C++ member), 232
  - `rtc_gpio_desc_t::slpie` (C++ member), 232
  - `rtc_gpio_desc_t::slpsel` (C++ member), 232
  - `rtc_gpio_force_hold_dis_all` (C++ function), 231
  - `rtc_gpio_get_drive_capability` (C++ function), 231
  - `rtc_gpio_get_level` (C++ function), 228
  - `rtc_gpio_hold_dis` (C++ function), 230
  - `rtc_gpio_hold_en` (C++ function), 230
  - `rtc_gpio_init` (C++ function), 228
  - `RTC_GPIO_IS_VALID_GPIO` (C macro), 232
  - `rtc_gpio_is_valid_gpio` (C++ function), 228
  - `RTC_GPIO_MODE_DISABLED` (C++ enumerator), 232
  - `RTC_GPIO_MODE_INPUT_ONLY` (C++ enumerator), 232
  - `RTC_GPIO_MODE_INPUT_OUTPUT` (C++ enumerator), 232
  - `RTC_GPIO_MODE_OUTPUT_ONLY` (C++ enumerator), 232
  - `rtc_gpio_mode_t` (C++ type), 232
  - `rtc_gpio_pulldown_dis` (C++ function), 230
  - `rtc_gpio_pulldown_en` (C++ function), 229
  - `rtc_gpio_pullup_dis` (C++ function), 229
  - `rtc_gpio_pullup_en` (C++ function), 229
  - `rtc_gpio_set_direction` (C++ function), 229
  - `rtc_gpio_set_drive_capability` (C++ function), 231
  - `rtc_gpio_set_level` (C++ function), 228
  - `RTC_SLOW_MEM` (C macro), 818
- ## S
- `sc_callback_t` (C++ type), 89
  - `SC_STATUS_FIND_CHANNEL` (C++ enumerator), 90
  - `SC_STATUS_GETTING_SSID_PSWD` (C++ enumerator), 90
  - `SC_STATUS_LINK` (C++ enumerator), 90
  - `SC_STATUS_LINK_OVER` (C++ enumerator), 90
  - `SC_STATUS_WAIT` (C++ enumerator), 90
  - `SC_TYPE_AIRKISS` (C++ enumerator), 90
  - `SC_TYPE_ESPTOUCH` (C++ enumerator), 90
  - `SC_TYPE_ESPTOUCH_AIRKISS` (C++ enumerator), 90
  - `sdmmc_card_init` (C++ function), 320
  - `sdmmc_card_t` (C++ class), 319
  - `sdmmc_card_t::cid` (C++ member), 319
  - `sdmmc_card_t::csd` (C++ member), 319
  - `sdmmc_card_t::host` (C++ member), 319
  - `sdmmc_card_t::ocr` (C++ member), 319
  - `sdmmc_card_t::rca` (C++ member), 319
  - `sdmmc_card_t::scr` (C++ member), 319
  - `sdmmc_cid_t` (C++ class), 320
  - `sdmmc_cid_t::date` (C++ member), 320
  - `sdmmc_cid_t::mfg_id` (C++ member), 320
  - `sdmmc_cid_t::name` (C++ member), 320
  - `sdmmc_cid_t::oem_id` (C++ member), 320
  - `sdmmc_cid_t::revision` (C++ member), 320
  - `sdmmc_cid_t::serial` (C++ member), 320
  - `sdmmc_command_t` (C++ class), 318
  - `sdmmc_command_t::arg` (C++ member), 319
  - `sdmmc_command_t::blklen` (C++ member), 319
  - `sdmmc_command_t::data` (C++ member), 319
  - `sdmmc_command_t::datalen` (C++ member), 319
  - `sdmmc_command_t::error` (C++ member), 319
  - `sdmmc_command_t::flags` (C++ member), 319
  - `sdmmc_command_t::opcode` (C++ member), 319
  - `sdmmc_command_t::response` (C++ member), 319
  - `sdmmc_command_t::timeout_ms` (C++ member), 319
  - `sdmmc_csd_t` (C++ class), 319
  - `sdmmc_csd_t::capacity` (C++ member), 320
  - `sdmmc_csd_t::card_command_class` (C++ member), 320
  - `sdmmc_csd_t::csd_ver` (C++ member), 320
  - `sdmmc_csd_t::mmc_ver` (C++ member), 320
  - `sdmmc_csd_t::read_block_len` (C++ member), 320
  - `sdmmc_csd_t::sector_size` (C++ member), 320
  - `sdmmc_csd_t::tr_speed` (C++ member), 320
  - `SDMMC_FREQ_DEFAULT` (C macro), 318
  - `SDMMC_FREQ_HIGHSPEED` (C macro), 318
  - `SDMMC_FREQ_PROBING` (C macro), 318
  - `SDMMC_HOST_DEFAULT` (C macro), 322
  - `sdmmc_host_deinit` (C++ function), 324
  - `sdmmc_host_do_transaction` (C++ function), 323
  - `SDMMC_HOST_FLAG_1BIT` (C macro), 318
  - `SDMMC_HOST_FLAG_4BIT` (C macro), 318
  - `SDMMC_HOST_FLAG_8BIT` (C macro), 318
  - `SDMMC_HOST_FLAG_SPI` (C macro), 318
  - `sdmmc_host_init` (C++ function), 322
  - `sdmmc_host_init_slot` (C++ function), 322
  - `sdmmc_host_set_bus_width` (C++ function), 323
  - `sdmmc_host_set_card_clk` (C++ function), 323
  - `SDMMC_HOST_SLOT_0` (C macro), 322
  - `SDMMC_HOST_SLOT_1` (C macro), 322
  - `sdmmc_host_t` (C++ class), 318

sdmmc\_host\_t::command\_timeout\_ms (C++ member), 318

sdmmc\_host\_t::deinit (C++ member), 318

sdmmc\_host\_t::do\_transaction (C++ member), 318

sdmmc\_host\_t::flags (C++ member), 318

sdmmc\_host\_t::init (C++ member), 318

sdmmc\_host\_t::io\_voltage (C++ member), 318

sdmmc\_host\_t::max\_freq\_khz (C++ member), 318

sdmmc\_host\_t::set\_bus\_width (C++ member), 318

sdmmc\_host\_t::set\_card\_clk (C++ member), 318

sdmmc\_host\_t::slot (C++ member), 318

sdmmc\_read\_sectors (C++ function), 321

sdmmc\_scr\_t (C++ class), 320

sdmmc\_scr\_t::bus\_width (C++ member), 320

sdmmc\_scr\_t::sd\_spec (C++ member), 320

SDMMC\_SLOT\_CONFIG\_DEFAULT (C macro), 323

sdmmc\_slot\_config\_t (C++ class), 322

sdmmc\_slot\_config\_t::gpio\_cd (C++ member), 323

sdmmc\_slot\_config\_t::gpio\_wp (C++ member), 323

sdmmc\_slot\_config\_t::width (C++ member), 323

SDMMC\_SLOT\_NO\_CD (C macro), 323

SDMMC\_SLOT\_NO\_WP (C macro), 323

SDMMC\_SLOT\_WIDTH\_DEFAULT (C macro), 322

sdmmc\_write\_sectors (C++ function), 321

SDSPI\_HOST\_DEFAULT (C macro), 325

sdspi\_host\_deinit (C++ function), 326

sdspi\_host\_do\_transaction (C++ function), 326

sdspi\_host\_init (C++ function), 324

sdspi\_host\_init\_slot (C++ function), 325

sdspi\_host\_set\_card\_clk (C++ function), 326

SDSPI\_SLOT\_CONFIG\_DEFAULT (C macro), 325

sdspi\_slot\_config\_t (C++ class), 325

sdspi\_slot\_config\_t::dma\_channel (C++ member), 325

sdspi\_slot\_config\_t::gpio\_cd (C++ member), 325

sdspi\_slot\_config\_t::gpio\_cs (C++ member), 325

sdspi\_slot\_config\_t::gpio\_miso (C++ member), 325

sdspi\_slot\_config\_t::gpio\_mosi (C++ member), 325

sdspi\_slot\_config\_t::gpio\_sck (C++ member), 325

sdspi\_slot\_config\_t::gpio\_wp (C++ member), 325

SDSPI\_SLOT\_NO\_CD (C macro), 325

SDSPI\_SLOT\_NO\_WP (C macro), 325

SemaphoreHandle\_t (C++ type), 511

semBINARY\_SEMAPHORE\_QUEUE\_LENGTH (C macro), 497

semGIVE\_BLOCK\_TIME (C macro), 497

semSEMAPHORE\_QUEUE\_ITEM\_LENGTH (C macro), 497

SIGMADELTA\_CHANNEL\_0 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_1 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_2 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_3 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_4 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_5 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_6 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_7 (C++ enumerator), 329

SIGMADELTA\_CHANNEL\_MAX (C++ enumerator), 329

sigmadelta\_channel\_t (C++ type), 329

sigmadelta\_config (C++ function), 327

sigmadelta\_config\_t (C++ class), 328

sigmadelta\_config\_t::channel (C++ member), 329

sigmadelta\_config\_t::sigmadelta\_duty (C++ member), 329

sigmadelta\_config\_t::sigmadelta\_gpio (C++ member), 329

sigmadelta\_config\_t::sigmadelta\_prescale (C++ member), 329

sigmadelta\_set\_duty (C++ function), 327

sigmadelta\_set\_pin (C++ function), 328

sigmadelta\_set\_prescale (C++ function), 328

slave\_transaction\_cb\_t (C++ type), 347

smartconfig\_status\_t (C++ type), 90

smartconfig\_type\_t (C++ type), 90

spi\_bus\_add\_device (C++ function), 337

spi\_bus\_config\_t (C++ class), 335

spi\_bus\_config\_t::max\_transfer\_sz (C++ member), 336

spi\_bus\_config\_t::miso\_io\_num (C++ member), 336

spi\_bus\_config\_t::mosi\_io\_num (C++ member), 336

spi\_bus\_config\_t::quadhd\_io\_num (C++ member), 336

spi\_bus\_config\_t::quadwp\_io\_num (C++ member), 336

spi\_bus\_config\_t::sclk\_io\_num (C++ member), 336

spi\_bus\_free (C++ function), 337

spi\_bus\_initialize (C++ function), 337

spi\_bus\_remove\_device (C++ function), 338

SPI\_DEVICE\_3WIRE (C macro), 341

SPI\_DEVICE\_BIT\_LSBFIRST (C macro), 341

SPI\_DEVICE\_CLK\_AS\_CS (C macro), 341

- `spi_device_get_trans_result` (C++ function), 338
- `SPI_DEVICE_HALFDUPLEX` (C macro), 341
- `spi_device_handle_t` (C++ type), 342
- `spi_device_interface_config_t` (C++ class), 339
- `spi_device_interface_config_t::address_bits` (C++ member), 339
- `spi_device_interface_config_t::clock_speed_hz` (C++ member), 340
- `spi_device_interface_config_t::command_bits` (C++ member), 339
- `spi_device_interface_config_t::cs_enable_pos` (C++ member), 340
- `spi_device_interface_config_t::cs_enable_pre` (C++ member), 340
- `spi_device_interface_config_t::dummy_bits` (C++ member), 339
- `spi_device_interface_config_t::duty_cycle` (C++ member), 340
- `spi_device_interface_config_t::flags` (C++ member), 340
- `spi_device_interface_config_t::mode` (C++ member), 340
- `spi_device_interface_config_t::post_cb` (C++ member), 340
- `spi_device_interface_config_t::pre_cb` (C++ member), 340
- `spi_device_interface_config_t::queue_size` (C++ member), 340
- `spi_device_interface_config_t::spics_io_num` (C++ member), 340
- `SPI_DEVICE_POSITIVE_CS` (C macro), 341
- `spi_device_queue_trans` (C++ function), 338
- `SPI_DEVICE_RXBIT_LSBFIRST` (C macro), 341
- `spi_device_transmit` (C++ function), 339
- `SPI_DEVICE_TXBIT_LSBFIRST` (C macro), 341
- `spi_flash_cache2phys` (C++ function), 408
- `SPI_FLASH_CACHE2PHYS_FAIL` (C macro), 410
- `spi_flash_cache_enabled` (C++ function), 409
- `spi_flash_erase_range` (C++ function), 406
- `spi_flash_erase_sector` (C++ function), 406
- `spi_flash_get_chip_size` (C++ function), 406
- `spi_flash_guard_end_func_t` (C++ type), 411
- `spi_flash_guard_funcs_t` (C++ class), 410
- `spi_flash_guard_funcs_t::end` (C++ member), 410
- `spi_flash_guard_funcs_t::op_lock` (C++ member), 410
- `spi_flash_guard_funcs_t::op_unlock` (C++ member), 410
- `spi_flash_guard_funcs_t::start` (C++ member), 410
- `spi_flash_guard_get` (C++ function), 409
- `spi_flash_guard_set` (C++ function), 409
- `spi_flash_guard_start_func_t` (C++ type), 411
- `spi_flash_init` (C++ function), 406
- `spi_flash_mmap` (C++ function), 407
- `SPI_FLASH_MMAP_DATA` (C++ enumerator), 411
- `spi_flash_mmap_dump` (C++ function), 408
- `spi_flash_mmap_get_free_pages` (C++ function), 408
- `spi_flash_mmap_handle_t` (C++ type), 411
- `SPI_FLASH_MMAP_INST` (C++ enumerator), 411
- `spi_flash_mmap_memory_t` (C++ type), 411
- `spi_flash_mmap_pages` (C++ function), 408
- `SPI_FLASH_MMU_PAGE_SIZE` (C macro), 410
- `spi_flash_munmap` (C++ function), 408
- `spi_flash_op_lock_func_t` (C++ type), 411
- `spi_flash_op_unlock_func_t` (C++ type), 411
- `spi_flash_phys2cache` (C++ function), 409
- `spi_flash_read` (C++ function), 407
- `spi_flash_read_encrypted` (C++ function), 407
- `SPI_FLASH_SEC_SIZE` (C macro), 410
- `spi_flash_write` (C++ function), 406
- `spi_flash_write_encrypted` (C++ function), 406
- `SPI_HOST` (C++ enumerator), 336
- `spi_host_device_t` (C++ type), 336
- `SPI_MAX_DMA_LEN` (C macro), 336
- `SPI_SLAVE_BIT_LSBFIRST` (C macro), 347
- `spi_slave_free` (C++ function), 344
- `spi_slave_get_trans_result` (C++ function), 345
- `spi_slave_initialize` (C++ function), 344
- `spi_slave_interface_config_t` (C++ class), 346
- `spi_slave_interface_config_t::flags` (C++ member), 346
- `spi_slave_interface_config_t::mode` (C++ member), 346
- `spi_slave_interface_config_t::post_setup_cb` (C++ member), 346
- `spi_slave_interface_config_t::post_trans_cb` (C++ member), 346
- `spi_slave_interface_config_t::queue_size` (C++ member), 346
- `spi_slave_interface_config_t::spics_io_num` (C++ member), 346
- `spi_slave_queue_trans` (C++ function), 344
- `SPI_SLAVE_RXBIT_LSBFIRST` (C macro), 346
- `spi_slave_transaction_t` (C++ class), 346
- `spi_slave_transaction_t` (C++ type), 347
- `spi_slave_transaction_t::length` (C++ member), 346
- `spi_slave_transaction_t::rx_buffer` (C++ member), 346

*spi\_slave\_transaction\_t::trans\_len* (C++ member), 346  
*spi\_slave\_transaction\_t::tx\_buffer* (C++ member), 346  
*spi\_slave\_transaction\_t::user* (C++ member), 346  
*spi\_slave\_transmit* (C++ function), 345  
*SPI\_SLAVE\_TXBIT\_LSBFIRST* (C macro), 346  
*SPI\_TRANS\_MODE\_DIO* (C macro), 341  
*SPI\_TRANS\_MODE\_DIOQIO\_ADDR* (C macro), 342  
*SPI\_TRANS\_MODE\_QIO* (C macro), 342  
*SPI\_TRANS\_USE\_RXDATA* (C macro), 342  
*SPI\_TRANS\_USE\_TXDATA* (C macro), 342  
*SPI\_TRANS\_VARIABLE\_ADDR* (C macro), 342  
*SPI\_TRANS\_VARIABLE\_CMD* (C macro), 342  
*spi\_transaction\_ext\_t* (C++ class), 341  
*spi\_transaction\_ext\_t::address\_bits* (C++ member), 341  
*spi\_transaction\_ext\_t::base* (C++ member), 341  
*spi\_transaction\_ext\_t::command\_bits* (C++ member), 341  
*spi\_transaction\_t* (C++ class), 340  
*spi\_transaction\_t* (C++ type), 342  
*spi\_transaction\_t::addr* (C++ member), 340  
*spi\_transaction\_t::cmd* (C++ member), 340  
*spi\_transaction\_t::flags* (C++ member), 340  
*spi\_transaction\_t::length* (C++ member), 341  
*spi\_transaction\_t::rx\_buffer* (C++ member), 341  
*spi\_transaction\_t::rx\_data* (C++ member), 341  
*spi\_transaction\_t::rxlength* (C++ member), 341  
*spi\_transaction\_t::tx\_buffer* (C++ member), 341  
*spi\_transaction\_t::tx\_data* (C++ member), 341  
*spi\_transaction\_t::user* (C++ member), 341  
*spicommon\_bus\_free\_io* (C++ function), 333  
*spicommon\_bus\_initialize\_io* (C++ function), 333  
*SPICOMMON\_BUSFLAG\_MASTER* (C macro), 336  
*SPICOMMON\_BUSFLAG\_QUAD* (C macro), 336  
*SPICOMMON\_BUSFLAG\_SLAVE* (C macro), 336  
*spicommon\_cs\_free* (C++ function), 333  
*spicommon\_cs\_initialize* (C++ function), 333  
*spicommon\_dma\_chan\_claim* (C++ function), 332  
*spicommon\_dma\_chan\_free* (C++ function), 332  
*spicommon\_dmaworkaround\_idle* (C++ function), 335  
*spicommon\_dmaworkaround\_req\_reset* (C++ function), 334  
*spicommon\_dmaworkaround\_reset\_in\_progress* (C++ function), 335  
*spicommon\_dmaworkaround\_transfer\_active* (C++ function), 335  
*spicommon\_freeze\_cs* (C++ function), 334  
*spicommon\_hw\_for\_host* (C++ function), 334  
*spicommon\_irqsource\_for\_host* (C++ function), 334  
*spicommon\_periph\_claim* (C++ function), 332  
*spicommon\_periph\_free* (C++ function), 332  
*spicommon\_restore\_cs* (C++ function), 334  
*spicommon\_setup\_dma\_desc\_links* (C++ function), 334

## T

*taskDISABLE\_INTERRUPTS* (C macro), 472  
*taskENABLE\_INTERRUPTS* (C macro), 473  
*taskENTER\_CRITICAL* (C macro), 472  
*taskENTER\_CRITICAL\_ISR* (C macro), 472  
*taskEXIT\_CRITICAL* (C macro), 472  
*taskEXIT\_CRITICAL\_ISR* (C macro), 472  
*TaskHandle\_t* (C++ type), 473  
*TaskHookFunction\_t* (C++ type), 473  
*taskSCHEDULER\_NOT\_STARTED* (C macro), 473  
*taskSCHEDULER\_RUNNING* (C macro), 473  
*taskSCHEDULER\_SUSPENDED* (C macro), 473  
*TaskSnapshot\_t* (C++ type), 474  
*TaskStatus\_t* (C++ type), 473  
*taskYIELD* (C macro), 472  
*TIMER\_0* (C++ enumerator), 355  
*TIMER\_1* (C++ enumerator), 355  
*TIMER\_ALARM\_DIS* (C++ enumerator), 355  
*TIMER\_ALARM\_EN* (C++ enumerator), 355  
*TIMER\_ALARM\_MAX* (C++ enumerator), 355  
*timer\_alarm\_t* (C++ type), 355  
*TIMER\_AUTORELOAD\_DIS* (C++ enumerator), 356  
*TIMER\_AUTORELOAD\_EN* (C++ enumerator), 356  
*TIMER\_AUTORELOAD\_MAX* (C++ enumerator), 356  
*timer\_autoreload\_t* (C++ type), 356  
*TIMER\_BASE\_CLK* (C macro), 354  
*timer\_config\_t* (C++ class), 354  
*timer\_config\_t::alarm\_en* (C++ member), 354  
*timer\_config\_t::auto\_reload* (C++ member), 354  
*timer\_config\_t::counter\_dir* (C++ member), 354  
*timer\_config\_t::counter\_en* (C++ member), 354  
*timer\_config\_t::divider* (C++ member), 354  
*timer\_config\_t::intr\_type* (C++ member), 354  
*timer\_count\_dir\_t* (C++ type), 355  
*TIMER\_COUNT\_DOWN* (C++ enumerator), 355  
*TIMER\_COUNT\_MAX* (C++ enumerator), 355

- TIMER\_COUNT\_UP (C++ enumerator), 355
- timer\_disable\_intr (C++ function), 353
- timer\_enable\_intr (C++ function), 353
- timer\_get\_alarm\_value (C++ function), 351
- timer\_get\_config (C++ function), 352
- timer\_get\_counter\_time\_sec (C++ function), 349
- timer\_get\_counter\_value (C++ function), 349
- TIMER\_GROUP\_0 (C++ enumerator), 355
- TIMER\_GROUP\_1 (C++ enumerator), 355
- timer\_group\_intr\_disable (C++ function), 353
- timer\_group\_intr\_enable (C++ function), 353
- TIMER\_GROUP\_MAX (C++ enumerator), 355
- timer\_group\_t (C++ type), 355
- timer\_idx\_t (C++ type), 355
- timer\_init (C++ function), 352
- TIMER\_INTR\_LEVEL (C++ enumerator), 356
- TIMER\_INTR\_MAX (C++ enumerator), 356
- timer\_intr\_mode\_t (C++ type), 355
- timer\_isr\_handle\_t (C++ type), 354
- timer\_isr\_register (C++ function), 352
- TIMER\_MAX (C++ enumerator), 355
- TIMER\_PAUSE (C++ enumerator), 355
- timer\_pause (C++ function), 350
- timer\_set\_alarm (C++ function), 351
- timer\_set\_alarm\_value (C++ function), 351
- timer\_set\_auto\_reload (C++ function), 350
- timer\_set\_counter\_mode (C++ function), 350
- timer\_set\_counter\_value (C++ function), 349
- timer\_set\_divider (C++ function), 351
- TIMER\_START (C++ enumerator), 355
- timer\_start (C++ function), 350
- timer\_start\_t (C++ type), 355
- TimerCallbackFunction\_t (C++ type), 529
- TimerHandle\_t (C++ type), 529
- TlsDeleteCallbackFunction\_t (C++ type), 474
- tmrCOMMAND\_CHANGE\_PERIOD (C macro), 519
- tmrCOMMAND\_CHANGE\_PERIOD\_FROM\_ISR (C macro), 519
- tmrCOMMAND\_DELETE (C macro), 519
- tmrCOMMAND\_EXECUTE\_CALLBACK (C macro), 519
- tmrCOMMAND\_EXECUTE\_CALLBACK\_FROM\_ISR (C macro), 519
- tmrCOMMAND\_RESET (C macro), 519
- tmrCOMMAND\_RESET\_FROM\_ISR (C macro), 519
- tmrCOMMAND\_START (C macro), 519
- tmrCOMMAND\_START\_DONT\_TRACE (C macro), 519
- tmrCOMMAND\_START\_FROM\_ISR (C macro), 519
- tmrCOMMAND\_STOP (C macro), 519
- tmrCOMMAND\_STOP\_FROM\_ISR (C macro), 519
- tmrFIRST\_FROM\_ISR\_COMMAND (C macro), 519
- touch\_cnt\_slope\_t (C++ type), 370
- TOUCH\_FSM\_MODE\_DEFAULT (C macro), 368
- TOUCH\_FSM\_MODE\_MAX (C++ enumerator), 371
- TOUCH\_FSM\_MODE\_SW (C++ enumerator), 371
- touch\_fsm\_mode\_t (C++ type), 370
- TOUCH\_FSM\_MODE\_TIMER (C++ enumerator), 371
- touch\_high\_volt\_t (C++ type), 369
- TOUCH\_HVOLT\_2V4 (C++ enumerator), 369
- TOUCH\_HVOLT\_2V5 (C++ enumerator), 369
- TOUCH\_HVOLT\_2V6 (C++ enumerator), 369
- TOUCH\_HVOLT\_2V7 (C++ enumerator), 369
- TOUCH\_HVOLT\_ATTEN\_0V (C++ enumerator), 369
- TOUCH\_HVOLT\_ATTEN\_0V5 (C++ enumerator), 369
- TOUCH\_HVOLT\_ATTEN\_1V (C++ enumerator), 369
- TOUCH\_HVOLT\_ATTEN\_1V5 (C++ enumerator), 369
- TOUCH\_HVOLT\_ATTEN\_KEEP (C++ enumerator), 369
- TOUCH\_HVOLT\_ATTEN\_MAX (C++ enumerator), 369
- TOUCH\_HVOLT\_KEEP (C++ enumerator), 369
- TOUCH\_HVOLT\_MAX (C++ enumerator), 369
- touch\_isr\_handle\_t (C++ type), 368
- touch\_low\_volt\_t (C++ type), 369
- TOUCH\_LVOLT\_0V5 (C++ enumerator), 369
- TOUCH\_LVOLT\_0V6 (C++ enumerator), 369
- TOUCH\_LVOLT\_0V7 (C++ enumerator), 369
- TOUCH\_LVOLT\_0V8 (C++ enumerator), 369
- TOUCH\_LVOLT\_KEEP (C++ enumerator), 369
- TOUCH\_LVOLT\_MAX (C++ enumerator), 369
- TOUCH\_PAD\_BIT\_MASK\_MAX (C macro), 368
- touch\_pad\_clear\_group\_mask (C++ function), 366
- touch\_pad\_clear\_status (C++ function), 366
- touch\_pad\_config (C++ function), 360
- touch\_pad\_deinit (C++ function), 360
- touch\_pad\_filter\_delete (C++ function), 367
- touch\_pad\_filter\_start (C++ function), 367
- touch\_pad\_filter\_stop (C++ function), 367
- touch\_pad\_get\_cnt\_mode (C++ function), 363
- touch\_pad\_get\_filter\_period (C++ function), 367
- touch\_pad\_get\_fsm\_mode (C++ function), 364
- touch\_pad\_get\_group\_mask (C++ function), 365
- touch\_pad\_get\_meas\_time (C++ function), 362
- touch\_pad\_get\_status (C++ function), 366
- touch\_pad\_get\_thresh (C++ function), 364
- touch\_pad\_get\_trigger\_mode (C++ function), 365
- touch\_pad\_get\_trigger\_source (C++ function), 365
- touch\_pad\_get\_voltage (C++ function), 362
- TOUCH\_PAD\_GPIO0\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO12\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO13\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO14\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO15\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO27\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO2\_CHANNEL (C macro), 371
- TOUCH\_PAD\_GPIO32\_CHANNEL (C macro), 371

- TOUCH\_PAD\_GPIO33\_CHANNEL (*C macro*), 371
- TOUCH\_PAD\_GPIO4\_CHANNEL (*C macro*), 371
- touch\_pad\_init (*C++ function*), 360
- touch\_pad\_intr\_disable (*C++ function*), 366
- touch\_pad\_intr\_enable (*C++ function*), 366
- touch\_pad\_io\_init (*C++ function*), 363
- touch\_pad\_isr\_deregister (*C++ function*), 361
- touch\_pad\_isr\_handler\_register (*C++ function*), 361
- touch\_pad\_isr\_register (*C++ function*), 361
- TOUCH\_PAD\_MAX (*C++ enumerator*), 369
- TOUCH\_PAD\_MEASURE\_CYCLE\_DEFAULT (*C macro*), 368
- TOUCH\_PAD\_NUM0 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM0\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM1 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM1\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM2 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM2\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM3 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM3\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM4 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM4\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM5 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM5\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM6 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM6\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM7 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM7\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM8 (*C++ enumerator*), 368
- TOUCH\_PAD\_NUM8\_GPIO\_NUM (*C macro*), 371
- TOUCH\_PAD\_NUM9 (*C++ enumerator*), 369
- TOUCH\_PAD\_NUM9\_GPIO\_NUM (*C macro*), 371
- touch\_pad\_read (*C++ function*), 360
- touch\_pad\_read\_filtered (*C++ function*), 360
- touch\_pad\_set\_cnt\_mode (*C++ function*), 363
- touch\_pad\_set\_filter\_period (*C++ function*), 366
- touch\_pad\_set\_fsm\_mode (*C++ function*), 363
- touch\_pad\_set\_group\_mask (*C++ function*), 365
- touch\_pad\_set\_meas\_time (*C++ function*), 361
- touch\_pad\_set\_thresh (*C++ function*), 364
- touch\_pad\_set\_trigger\_mode (*C++ function*), 364
- touch\_pad\_set\_trigger\_source (*C++ function*), 365
- touch\_pad\_set\_voltage (*C++ function*), 362
- TOUCH\_PAD\_SLEEP\_CYCLE\_DEFAULT (*C macro*), 368
- TOUCH\_PAD\_SLOPE\_0 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_1 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_2 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_3 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_4 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_5 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_6 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_7 (*C++ enumerator*), 370
- TOUCH\_PAD\_SLOPE\_MAX (*C++ enumerator*), 370
- touch\_pad\_sw\_start (*C++ function*), 364
- touch\_pad\_t (*C++ type*), 368
- TOUCH\_PAD\_TIE\_OPT\_HIGH (*C++ enumerator*), 370
- TOUCH\_PAD\_TIE\_OPT\_LOW (*C++ enumerator*), 370
- TOUCH\_PAD\_TIE\_OPT\_MAX (*C++ enumerator*), 370
- touch\_tie\_opt\_t (*C++ type*), 370
- TOUCH\_TRIGGER\_ABOVE (*C++ enumerator*), 370
- TOUCH\_TRIGGER\_BELOW (*C++ enumerator*), 370
- TOUCH\_TRIGGER\_MAX (*C++ enumerator*), 370
- TOUCH\_TRIGGER\_MODE\_DEFAULT (*C macro*), 368
- touch\_trigger\_mode\_t (*C++ type*), 370
- TOUCH\_TRIGGER\_SOURCE\_BOTH (*C++ enumerator*), 370
- TOUCH\_TRIGGER\_SOURCE\_DEFAULT (*C macro*), 368
- TOUCH\_TRIGGER\_SOURCE\_MAX (*C++ enumerator*), 370
- TOUCH\_TRIGGER\_SOURCE\_SET1 (*C++ enumerator*), 370
- touch\_trigger\_src\_t (*C++ type*), 370
- touch\_volt\_atten\_t (*C++ type*), 369
- transaction\_cb\_t (*C++ type*), 342
- tskIDLE\_PRIORITY (*C macro*), 472
- tskKERNEL\_VERSION\_BUILD (*C macro*), 472
- tskKERNEL\_VERSION\_MAJOR (*C macro*), 472
- tskKERNEL\_VERSION\_MINOR (*C macro*), 472
- tskKERNEL\_VERSION\_NUMBER (*C macro*), 472
- tskNO\_AFFINITY (*C macro*), 472
- ## U
- UART\_BITRATE\_MAX (*C macro*), 387
- UART\_BREAK (*C++ enumerator*), 389
- UART\_BUFFER\_FULL (*C++ enumerator*), 389
- uart\_clear\_intr\_status (*C++ function*), 378
- uart\_config\_t (*C++ class*), 386
- uart\_config\_t::baud\_rate (*C++ member*), 386
- uart\_config\_t::data\_bits (*C++ member*), 386
- uart\_config\_t::flow\_ctrl (*C++ member*), 387
- uart\_config\_t::parity (*C++ member*), 386
- uart\_config\_t::rx\_flow\_ctrl\_thresh (*C++ member*), 387
- uart\_config\_t::stop\_bits (*C++ member*), 386
- uart\_config\_t::use\_ref\_tick (*C++ member*), 387
- UART\_CTS\_GPIO19\_DIRECT\_CHANNEL (*C macro*), 390
- UART\_CTS\_GPIO6\_DIRECT\_CHANNEL (*C macro*), 391
- UART\_CTS\_GPIO8\_DIRECT\_CHANNEL (*C macro*), 391
- UART\_DATA (*C++ enumerator*), 389

- UART\_DATA\_5\_BITS (C++ enumerator), 388
- UART\_DATA\_6\_BITS (C++ enumerator), 388
- UART\_DATA\_7\_BITS (C++ enumerator), 388
- UART\_DATA\_8\_BITS (C++ enumerator), 388
- UART\_DATA\_BITS\_MAX (C++ enumerator), 388
- UART\_DATA\_BREAK (C++ enumerator), 390
- uart\_disable\_intr\_mask (C++ function), 378
- uart\_disable\_pattern\_det\_intr (C++ function), 385
- uart\_disable\_rx\_intr (C++ function), 379
- uart\_disable\_tx\_intr (C++ function), 379
- uart\_driver\_delete (C++ function), 382
- uart\_driver\_install (C++ function), 382
- uart\_enable\_intr\_mask (C++ function), 378
- uart\_enable\_pattern\_det\_intr (C++ function), 385
- uart\_enable\_rx\_intr (C++ function), 379
- uart\_enable\_tx\_intr (C++ function), 379
- UART\_EVENT\_MAX (C++ enumerator), 390
- uart\_event\_t (C++ class), 387
- uart\_event\_t::size (C++ member), 387
- uart\_event\_t::type (C++ member), 387
- uart\_event\_type\_t (C++ type), 389
- UART\_FIFO\_LEN (C macro), 387
- UART\_FIFO\_OVF (C++ enumerator), 389
- uart\_flush (C++ function), 384
- uart\_flush\_input (C++ function), 384
- UART\_FRAME\_ERR (C++ enumerator), 389
- uart\_get\_baudrate (C++ function), 376
- uart\_get\_buffered\_data\_len (C++ function), 384
- uart\_get\_hw\_flow\_ctrl (C++ function), 378
- uart\_get\_parity (C++ function), 376
- uart\_get\_stop\_bits (C++ function), 376
- uart\_get\_word\_length (C++ function), 375
- UART\_GPIO10\_DIRECT\_CHANNEL (C macro), 390
- UART\_GPIO11\_DIRECT\_CHANNEL (C macro), 391
- UART\_GPIO16\_DIRECT\_CHANNEL (C macro), 391
- UART\_GPIO17\_DIRECT\_CHANNEL (C macro), 391
- UART\_GPIO19\_DIRECT\_CHANNEL (C macro), 390
- UART\_GPIO1\_DIRECT\_CHANNEL (C macro), 390
- UART\_GPIO22\_DIRECT\_CHANNEL (C macro), 390
- UART\_GPIO3\_DIRECT\_CHANNEL (C macro), 390
- UART\_GPIO6\_DIRECT\_CHANNEL (C macro), 391
- UART\_GPIO7\_DIRECT\_CHANNEL (C macro), 391
- UART\_GPIO8\_DIRECT\_CHANNEL (C macro), 391
- UART\_GPIO9\_DIRECT\_CHANNEL (C macro), 390
- uart\_hw\_flowcontrol\_t (C++ type), 389
- UART\_HW\_FLOWCTRL\_CTS (C++ enumerator), 389
- UART\_HW\_FLOWCTRL\_CTS\_RTS (C++ enumerator), 389
- UART\_HW\_FLOWCTRL\_DISABLE (C++ enumerator), 389
- UART\_HW\_FLOWCTRL\_MAX (C++ enumerator), 389
- UART\_HW\_FLOWCTRL\_RTS (C++ enumerator), 389
- uart\_intr\_config (C++ function), 381
- uart\_intr\_config\_t (C++ class), 387
- uart\_intr\_config\_t::intr\_enable\_mask (C++ member), 387
- uart\_intr\_config\_t::rx\_timeout\_thresh (C++ member), 387
- uart\_intr\_config\_t::rxfifo\_full\_thresh (C++ member), 387
- uart\_intr\_config\_t::txfifo\_empty\_intr\_thresh (C++ member), 387
- UART\_INTR\_MASK (C macro), 387
- UART\_INVERSE\_CTS (C macro), 388
- UART\_INVERSE\_DISABLE (C macro), 387
- UART\_INVERSE\_RTS (C macro), 388
- UART\_INVERSE\_RXD (C macro), 388
- UART\_INVERSE\_TXD (C macro), 388
- uart\_isr\_free (C++ function), 380
- uart\_isr\_handle\_t (C++ type), 388
- uart\_isr\_register (C++ function), 379
- UART\_LINE\_INV\_MASK (C macro), 387
- UART\_NUM\_0 (C++ enumerator), 388
- UART\_NUM\_0\_CTS\_DIRECT\_GPIO\_NUM (C macro), 390
- UART\_NUM\_0\_RTS\_DIRECT\_GPIO\_NUM (C macro), 390
- UART\_NUM\_0\_RXD\_DIRECT\_GPIO\_NUM (C macro), 390
- UART\_NUM\_0\_TXD\_DIRECT\_GPIO\_NUM (C macro), 390
- UART\_NUM\_1 (C++ enumerator), 389
- UART\_NUM\_1\_CTS\_DIRECT\_GPIO\_NUM (C macro), 391
- UART\_NUM\_1\_RTS\_DIRECT\_GPIO\_NUM (C macro), 391
- UART\_NUM\_1\_RXD\_DIRECT\_GPIO\_NUM (C macro), 390
- UART\_NUM\_1\_TXD\_DIRECT\_GPIO\_NUM (C macro), 390
- UART\_NUM\_2 (C++ enumerator), 389
- UART\_NUM\_2\_CTS\_DIRECT\_GPIO\_NUM (C macro), 391
- UART\_NUM\_2\_RTS\_DIRECT\_GPIO\_NUM (C macro), 391
- UART\_NUM\_2\_RXD\_DIRECT\_GPIO\_NUM (C macro), 391
- UART\_NUM\_2\_TXD\_DIRECT\_GPIO\_NUM (C macro), 391
- UART\_NUM\_MAX (C++ enumerator), 389
- uart\_param\_config (C++ function), 381
- UART\_PARITY\_DISABLE (C++ enumerator), 389
- UART\_PARITY\_ERR (C++ enumerator), 389
- UART\_PARITY\_EVEN (C++ enumerator), 389
- UART\_PARITY\_ODD (C++ enumerator), 389

[uart\\_parity\\_t \(C++ type\), 389](#)  
[UART\\_PATTERN\\_DET \(C++ enumerator\), 390](#)  
[uart\\_pattern\\_get\\_pos \(C++ function\), 386](#)  
[uart\\_pattern\\_pop\\_pos \(C++ function\), 385](#)  
[uart\\_pattern\\_queue\\_reset \(C++ function\), 386](#)  
[UART\\_PIN\\_NO\\_CHANGE \(C macro\), 387](#)  
[uart\\_port\\_t \(C++ type\), 388](#)  
[uart\\_read\\_bytes \(C++ function\), 384](#)  
[UART\\_RTS\\_GPIO11\\_DIRECT\\_CHANNEL \(C macro\), 391](#)  
[UART\\_RTS\\_GPIO22\\_DIRECT\\_CHANNEL \(C macro\), 390](#)  
[UART\\_RTS\\_GPIO7\\_DIRECT\\_CHANNEL \(C macro\), 391](#)  
[UART\\_RXD\\_GPIO16\\_DIRECT\\_CHANNEL \(C macro\), 391](#)  
[UART\\_RXD\\_GPIO3\\_DIRECT\\_CHANNEL \(C macro\), 390](#)  
[UART\\_RXD\\_GPIO9\\_DIRECT\\_CHANNEL \(C macro\), 391](#)  
[uart\\_set\\_baudrate \(C++ function\), 376](#)  
[uart\\_set\\_dtr \(C++ function\), 381](#)  
[uart\\_set\\_hw\\_flow\\_ctrl \(C++ function\), 377](#)  
[uart\\_set\\_line\\_inverse \(C++ function\), 377](#)  
[uart\\_set\\_parity \(C++ function\), 376](#)  
[uart\\_set\\_pin \(C++ function\), 380](#)  
[uart\\_set\\_rts \(C++ function\), 380](#)  
[uart\\_set\\_stop\\_bits \(C++ function\), 375](#)  
[uart\\_set\\_sw\\_flow\\_ctrl \(C++ function\), 377](#)  
[uart\\_set\\_tx\\_idle\\_num \(C++ function\), 381](#)  
[uart\\_set\\_word\\_length \(C++ function\), 375](#)  
[UART\\_STOP\\_BITS\\_1 \(C++ enumerator\), 388](#)  
[UART\\_STOP\\_BITS\\_1\\_5 \(C++ enumerator\), 388](#)  
[UART\\_STOP\\_BITS\\_2 \(C++ enumerator\), 388](#)  
[UART\\_STOP\\_BITS\\_MAX \(C++ enumerator\), 388](#)  
[uart\\_stop\\_bits\\_t \(C++ type\), 388](#)  
[uart\\_tx\\_chars \(C++ function\), 382](#)  
[UART\\_TXD\\_GPIO10\\_DIRECT\\_CHANNEL \(C macro\), 391](#)  
[UART\\_TXD\\_GPIO17\\_DIRECT\\_CHANNEL \(C macro\), 391](#)  
[UART\\_TXD\\_GPIO1\\_DIRECT\\_CHANNEL \(C macro\), 390](#)  
[uart\\_wait\\_tx\\_done \(C++ function\), 382](#)  
[uart\\_word\\_length\\_t \(C++ type\), 388](#)  
[uart\\_write\\_bytes \(C++ function\), 383](#)  
[uart\\_write\\_bytes\\_with\\_break \(C++ function\), 383](#)  
[ulp\\_load\\_binary \(C++ function\), 820](#)  
[ulp\\_process\\_macros\\_and\\_load \(C++ function\), 814](#)  
[ulp\\_run \(C++ function\), 814, 821](#)  
[ulp\\_set\\_wakeup\\_period \(C++ function\), 821](#)  
[ulTaskNotifyTake \(C++ function\), 470](#)

[uxQueueMessagesWaiting \(C++ function\), 479](#)  
[uxQueueMessagesWaitingFromISR \(C++ function\), 476](#)  
[uxQueueSpacesAvailable \(C++ function\), 479](#)  
[uxSemaphoreGetCount \(C macro\), 511](#)  
[uxTaskGetNumberOfTasks \(C++ function\), 461](#)  
[uxTaskGetStackHighWaterMark \(C++ function\), 461](#)  
[uxTaskGetSystemState \(C++ function\), 463](#)  
[uxTaskPriorityGet \(C++ function\), 455](#)  
[uxTaskPriorityGetFromISR \(C++ function\), 456](#)

## V

[vendor\\_ie\\_data\\_t \(C++ class\), 81](#)  
[vendor\\_ie\\_data\\_t::element\\_id \(C++ member\), 81](#)  
[vendor\\_ie\\_data\\_t::length \(C++ member\), 81](#)  
[vendor\\_ie\\_data\\_t::payload \(C++ member\), 82](#)  
[vendor\\_ie\\_data\\_t::vendor\\_oui \(C++ member\), 82](#)  
[vendor\\_ie\\_data\\_t::vendor\\_oui\\_type \(C++ member\), 82](#)  
[vEventGroupDelete \(C++ function\), 536](#)  
[vprintf\\_like\\_t \(C++ type\), 587](#)  
[vQueueAddToRegistry \(C++ function\), 481](#)  
[vQueueDelete \(C++ function\), 479](#)  
[vQueueUnregisterQueue \(C++ function\), 481](#)  
[vRingbufferDelete \(C++ function\), 539](#)  
[vRingbufferReturnItem \(C++ function\), 542](#)  
[vRingbufferReturnItemFromISR \(C++ function\), 542](#)  
[vSemaphoreDelete \(C macro\), 510](#)  
[VSPI\\_HOST \(C++ enumerator\), 336](#)  
[vTaskDelay \(C++ function\), 454](#)  
[vTaskDelayUntil \(C++ function\), 455](#)  
[vTaskDelete \(C++ function\), 453](#)  
[vTaskGetRunTimeStats \(C++ function\), 466](#)  
[vTaskList \(C++ function\), 465](#)  
[vTaskNotifyGiveFromISR \(C++ function\), 469](#)  
[vTaskPrioritySet \(C++ function\), 456](#)  
[vTaskResume \(C++ function\), 458](#)  
[vTaskSetApplicationTaskTag \(C++ function\), 461](#)  
[vTaskSetThreadLocalStoragePointer \(C++ function\), 462](#)  
[vTaskSetThreadLocalStoragePointerAndDelCallback \(C++ function\), 462](#)  
[vTaskSuspend \(C++ function\), 457](#)  
[vTaskSuspendAll \(C++ function\), 459](#)  
[vTimerSetTimerID \(C++ function\), 516](#)

## W

[wifi\\_active\\_scan\\_time\\_t \(C++ class\), 78](#)

- wifi\_active\_scan\_time\_t::max (C++ member), 79
- wifi\_active\_scan\_time\_t::min (C++ member), 79
- WIFI\_ALL\_CHANNEL\_SCAN (C++ enumerator), 87
- WIFI\_AMPDU\_RX\_ENABLED (C macro), 77
- WIFI\_AMPDU\_TX\_ENABLED (C macro), 77
- wifi\_ap\_config\_t (C++ class), 80
- wifi\_ap\_config\_t::authmode (C++ member), 80
- wifi\_ap\_config\_t::beacon\_interval (C++ member), 80
- wifi\_ap\_config\_t::channel (C++ member), 80
- wifi\_ap\_config\_t::max\_connection (C++ member), 80
- wifi\_ap\_config\_t::password (C++ member), 80
- wifi\_ap\_config\_t::ssid (C++ member), 80
- wifi\_ap\_config\_t::ssid\_hidden (C++ member), 80
- wifi\_ap\_config\_t::ssid\_len (C++ member), 80
- wifi\_ap\_record\_t (C++ class), 79
- wifi\_ap\_record\_t::authmode (C++ member), 79
- wifi\_ap\_record\_t::bssid (C++ member), 79
- wifi\_ap\_record\_t::group\_cipher (C++ member), 79
- wifi\_ap\_record\_t::pairwise\_cipher (C++ member), 79
- wifi\_ap\_record\_t::phy\_11b (C++ member), 79
- wifi\_ap\_record\_t::phy\_11g (C++ member), 80
- wifi\_ap\_record\_t::phy\_11n (C++ member), 80
- wifi\_ap\_record\_t::phy\_lr (C++ member), 80
- wifi\_ap\_record\_t::primary (C++ member), 79
- wifi\_ap\_record\_t::reserved (C++ member), 80
- wifi\_ap\_record\_t::rssi (C++ member), 79
- wifi\_ap\_record\_t::second (C++ member), 79
- wifi\_ap\_record\_t::ssid (C++ member), 79
- wifi\_ap\_record\_t::wps (C++ member), 80
- WIFI\_AUTH\_MAX (C++ enumerator), 85
- wifi\_auth\_mode\_t (C++ type), 85
- WIFI\_AUTH\_OPEN (C++ enumerator), 85
- WIFI\_AUTH\_WEP (C++ enumerator), 85
- WIFI\_AUTH\_WPA2\_ENTERPRISE (C++ enumerator), 85
- WIFI\_AUTH\_WPA2\_PSK (C++ enumerator), 85
- WIFI\_AUTH\_WPA\_PSK (C++ enumerator), 85
- WIFI\_AUTH\_WPA\_WPA2\_PSK (C++ enumerator), 85
- wifi\_bandwidth\_t (C++ type), 87
- WIFI\_BW\_HT20 (C++ enumerator), 87
- WIFI\_BW\_HT40 (C++ enumerator), 87
- WIFI\_CIPHER\_TYPE\_CCMP (C++ enumerator), 86
- WIFI\_CIPHER\_TYPE\_NONE (C++ enumerator), 86
- wifi\_cipher\_type\_t (C++ type), 86
- WIFI\_CIPHER\_TYPE\_TKIP (C++ enumerator), 86
- WIFI\_CIPHER\_TYPE\_TKIP\_CCMP (C++ enumerator), 86
- WIFI\_CIPHER\_TYPE\_UNKNOWN (C++ enumerator), 86
- WIFI\_CIPHER\_TYPE\_WEP104 (C++ enumerator), 86
- WIFI\_CIPHER\_TYPE\_WEP40 (C++ enumerator), 86
- wifi\_config\_t (C++ type), 78
- wifi\_config\_t::ap (C++ member), 78
- wifi\_config\_t::sta (C++ member), 78
- WIFI\_CONNECT\_AP\_BY\_SECURITY (C++ enumerator), 87
- WIFI\_CONNECT\_AP\_BY\_SIGNAL (C++ enumerator), 87
- WIFI\_COUNTRY\_POLICY\_AUTO (C++ enumerator), 84
- WIFI\_COUNTRY\_POLICY\_MANUAL (C++ enumerator), 84
- wifi\_country\_policy\_t (C++ type), 84
- wifi\_country\_t (C++ class), 78
- wifi\_country\_t::cc (C++ member), 78
- wifi\_country\_t::nchan (C++ member), 78
- wifi\_country\_t::policy (C++ member), 78
- wifi\_country\_t::schan (C++ member), 78
- WIFI\_DEFAULT\_RX\_BA\_WIN (C macro), 77
- WIFI\_DEFAULT\_TX\_BA\_WIN (C macro), 77
- WIFI\_DYNAMIC\_TX\_BUFFER\_NUM (C macro), 77
- wifi\_err\_reason\_t (C++ type), 85
- WIFI\_FAST\_SCAN (C++ enumerator), 86
- wifi\_fast\_scan\_threshold\_t (C++ class), 80
- wifi\_fast\_scan\_threshold\_t::authmode (C++ member), 80
- wifi\_fast\_scan\_threshold\_t::rssi (C++ member), 80
- WIFI\_IF\_AP (C macro), 83
- WIFI\_IF\_STA (C macro), 83
- WIFI\_INIT\_CONFIG\_DEFAULT (C macro), 77
- WIFI\_INIT\_CONFIG\_MAGIC (C macro), 77
- wifi\_init\_config\_t (C++ class), 75
- wifi\_init\_config\_t::ampdu\_rx\_enable (C++ member), 75
- wifi\_init\_config\_t::ampdu\_tx\_enable (C++ member), 75
- wifi\_init\_config\_t::dynamic\_rx\_buf\_num (C++ member), 75
- wifi\_init\_config\_t::dynamic\_tx\_buf\_num (C++ member), 75
- wifi\_init\_config\_t::event\_handler (C++ member), 75
- wifi\_init\_config\_t::magic (C++ member), 76

*wifi\_init\_config\_t::nano\_enable* (C++ member), 75  
*wifi\_init\_config\_t::nvs\_enable* (C++ member), 75  
*wifi\_init\_config\_t::rx\_ba\_win* (C++ member), 76  
*wifi\_init\_config\_t::static\_rx\_buf\_num* (C++ member), 75  
*wifi\_init\_config\_t::static\_tx\_buf\_num* (C++ member), 75  
*wifi\_init\_config\_t::tx\_ba\_win* (C++ member), 75  
*wifi\_init\_config\_t::tx\_buf\_type* (C++ member), 75  
*wifi\_init\_config\_t::wpa\_crypto\_funcs* (C++ member), 75  
*wifi\_interface\_t* (C++ type), 84  
WIFI\_MODE\_AP (C++ enumerator), 84  
WIFI\_MODE\_APSTA (C++ enumerator), 84  
WIFI\_MODE\_MAX (C++ enumerator), 84  
WIFI\_MODE\_NULL (C++ enumerator), 84  
WIFI\_MODE\_STA (C++ enumerator), 84  
*wifi\_mode\_t* (C++ type), 84  
WIFI\_NANO\_FORMAT\_ENABLED (C macro), 77  
WIFI\_NVS\_ENABLED (C macro), 77  
WIFI\_PKT\_DATA (C++ enumerator), 88  
WIFI\_PKT\_MGMT (C++ enumerator), 88  
WIFI\_PKT\_MISC (C++ enumerator), 88  
*wifi\_pkt\_rx\_ctrl\_t* (C++ class), 82  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad0\_\_* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad1\_\_* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad2\_\_* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad3\_\_* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad4\_\_* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad5\_\_* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad6\_\_* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::\_\_pad7\_\_* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::aggregation* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::ampdu\_cnt* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::channel* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::cwb* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::fec\_coding* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::mcs* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::noise\_floor* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::not\_sounding* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::rate* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::rssi* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::rx\_state* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::sgi* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::sig\_len* (C++ member), 83  
*wifi\_pkt\_rx\_ctrl\_t::sig\_mode* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::smoothing* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::stbc* (C++ member), 82  
*wifi\_pkt\_rx\_ctrl\_t::timestamp* (C++ member), 83  
WIFI\_PROMIS\_FILTER\_MASK\_ALL (C macro), 84  
WIFI\_PROMIS\_FILTER\_MASK\_DATA (C macro), 84  
WIFI\_PROMIS\_FILTER\_MASK\_DATA\_AMPDU (C macro), 84  
WIFI\_PROMIS\_FILTER\_MASK\_DATA\_MPDU (C macro), 84  
WIFI\_PROMIS\_FILTER\_MASK\_MGMT (C macro), 84  
WIFI\_PROMIS\_FILTER\_MASK\_MISC (C macro), 84  
*wifi\_promiscuous\_cb\_t* (C++ type), 77  
*wifi\_promiscuous\_filter\_t* (C++ class), 83  
*wifi\_promiscuous\_filter\_t::filter\_mask* (C++ member), 83  
*wifi\_promiscuous\_pkt\_t* (C++ class), 83  
*wifi\_promiscuous\_pkt\_t::payload* (C++ member), 83  
*wifi\_promiscuous\_pkt\_t::rx\_ctrl* (C++ member), 83  
*wifi\_promiscuous\_pkt\_type\_t* (C++ type), 87  
WIFI\_PROTOCOL\_11B (C macro), 83  
WIFI\_PROTOCOL\_11G (C macro), 83  
WIFI\_PROTOCOL\_11N (C macro), 83  
WIFI\_PROTOCOL\_LR (C macro), 84  
WIFI\_PS\_MODEM (C++ enumerator), 87  
WIFI\_PS\_NONE (C++ enumerator), 87  
*wifi\_ps\_type\_t* (C++ type), 87  
WIFI\_REASON\_4WAY\_HANDSHAKE\_TIMEOUT (C++ enumerator), 85  
WIFI\_REASON\_802\_1X\_AUTH\_FAILED (C++ enumerator), 86  
WIFI\_REASON\_AKMP\_INVALID (C++ enumerator), 85  
WIFI\_REASON\_ASSOC\_EXPIRE (C++ enumerator), 85  
WIFI\_REASON\_ASSOC\_FAIL (C++ enumerator), 86  
WIFI\_REASON\_ASSOC\_LEAVE (C++ enumerator),

- 85
- WIFI\_REASON\_ASSOC\_NOT\_AUTHED (C++ *enumerator*), 85
- WIFI\_REASON\_ASSOC\_TOOMANY (C++ *enumerator*), 85
- WIFI\_REASON\_AUTH\_EXPIRE (C++ *enumerator*), 85
- WIFI\_REASON\_AUTH\_FAIL (C++ *enumerator*), 86
- WIFI\_REASON\_AUTH\_LEAVE (C++ *enumerator*), 85
- WIFI\_REASON\_BEACON\_TIMEOUT (C++ *enumerator*), 86
- WIFI\_REASON\_CIPHER\_SUITE\_REJECTED (C++ *enumerator*), 86
- WIFI\_REASON\_CONNECTION\_FAIL (C++ *enumerator*), 86
- WIFI\_REASON\_DISASSOC\_PWRCAP\_BAD (C++ *enumerator*), 85
- WIFI\_REASON\_DISASSOC\_SUPCHAN\_BAD (C++ *enumerator*), 85
- WIFI\_REASON\_GROUP\_CIPHER\_INVALID (C++ *enumerator*), 85
- WIFI\_REASON\_GROUP\_KEY\_UPDATE\_TIMEOUT (C++ *enumerator*), 85
- WIFI\_REASON\_HANDSHAKE\_TIMEOUT (C++ *enumerator*), 86
- WIFI\_REASON\_IE\_IN\_4WAY\_DIFFERS (C++ *enumerator*), 85
- WIFI\_REASON\_IE\_INVALID (C++ *enumerator*), 85
- WIFI\_REASON\_INVALID\_RSN\_IE\_CAP (C++ *enumerator*), 85
- WIFI\_REASON\_MIC\_FAILURE (C++ *enumerator*), 85
- WIFI\_REASON\_NO\_AP\_FOUND (C++ *enumerator*), 86
- WIFI\_REASON\_NOT\_ASSOCED (C++ *enumerator*), 85
- WIFI\_REASON\_NOT\_AUTHED (C++ *enumerator*), 85
- WIFI\_REASON\_PAIRWISE\_CIPHER\_INVALID (C++ *enumerator*), 85
- WIFI\_REASON\_UNSPECIFIED (C++ *enumerator*), 85
- WIFI\_REASON\_UNSUPP\_RSN\_IE\_VERSION (C++ *enumerator*), 85
- wifi\_scan\_config\_t (C++ *class*), 79
- wifi\_scan\_config\_t::bssid (C++ *member*), 79
- wifi\_scan\_config\_t::channel (C++ *member*), 79
- wifi\_scan\_config\_t::scan\_time (C++ *member*), 79
- wifi\_scan\_config\_t::scan\_type (C++ *member*), 79
- wifi\_scan\_config\_t::show\_hidden (C++ *member*), 79
- wifi\_scan\_config\_t::ssid (C++ *member*), 79
- wifi\_scan\_method\_t (C++ *type*), 86
- wifi\_scan\_threshold\_t (C++ *type*), 84
- wifi\_scan\_time\_t (C++ *type*), 78
- wifi\_scan\_time\_t::active (C++ *member*), 78
- wifi\_scan\_time\_t::passive (C++ *member*), 78
- WIFI\_SCAN\_TYPE\_ACTIVE (C++ *enumerator*), 86
- WIFI\_SCAN\_TYPE\_PASSIVE (C++ *enumerator*), 86
- wifi\_scan\_type\_t (C++ *type*), 86
- WIFI\_SECOND\_CHAN\_ABOVE (C++ *enumerator*), 86
- WIFI\_SECOND\_CHAN\_BELOW (C++ *enumerator*), 86
- WIFI\_SECOND\_CHAN\_NONE (C++ *enumerator*), 86
- wifi\_second\_chan\_t (C++ *type*), 86
- wifi\_sort\_method\_t (C++ *type*), 87
- wifi\_sta\_config\_t (C++ *class*), 80
- wifi\_sta\_config\_t::bssid (C++ *member*), 81
- wifi\_sta\_config\_t::bssid\_set (C++ *member*), 81
- wifi\_sta\_config\_t::channel (C++ *member*), 81
- wifi\_sta\_config\_t::password (C++ *member*), 81
- wifi\_sta\_config\_t::scan\_method (C++ *member*), 81
- wifi\_sta\_config\_t::sort\_method (C++ *member*), 81
- wifi\_sta\_config\_t::ssid (C++ *member*), 81
- wifi\_sta\_config\_t::threshold (C++ *member*), 81
- wifi\_sta\_info\_t (C++ *class*), 81
- wifi\_sta\_info\_t::mac (C++ *member*), 81
- wifi\_sta\_list\_t (C++ *class*), 81
- wifi\_sta\_list\_t::num (C++ *member*), 81
- wifi\_sta\_list\_t::sta (C++ *member*), 81
- WIFI\_STATIC\_TX\_BUFFER\_NUM (C *macro*), 77
- WIFI\_STORAGE\_FLASH (C++ *enumerator*), 87
- WIFI\_STORAGE\_RAM (C++ *enumerator*), 87
- wifi\_storage\_t (C++ *type*), 87
- WIFI\_VENDOR\_IE\_ELEMENT\_ID (C *macro*), 84
- wifi\_vendor\_ie\_id\_t (C++ *type*), 87
- wifi\_vendor\_ie\_type\_t (C++ *type*), 87
- WIFI\_VND\_IE\_ID\_0 (C++ *enumerator*), 87
- WIFI\_VND\_IE\_ID\_1 (C++ *enumerator*), 87
- WIFI\_VND\_IE\_TYPE\_ASSOC\_REQ (C++ *enumerator*), 87
- WIFI\_VND\_IE\_TYPE\_ASSOC\_RESP (C++ *enumerator*), 87
- WIFI\_VND\_IE\_TYPE\_BEACON (C++ *enumerator*), 87
- WIFI\_VND\_IE\_TYPE\_PROBE\_REQ (C++ *enumerator*), 87
- WIFI\_VND\_IE\_TYPE\_PROBE\_RESP (C++ *enumerator*), 87
- wl\_erase\_range (C++ *function*), 444
- wl\_handle\_t (C++ *type*), 446

WL\_INVALID\_HANDLE (*C macro*), 446  
 wl\_mount (*C++ function*), 444  
 wl\_read (*C++ function*), 445  
 wl\_sector\_size (*C++ function*), 446  
 wl\_size (*C++ function*), 446  
 wl\_unmount (*C++ function*), 444  
 wl\_write (*C++ function*), 445

## X

xEventGroupClearBits (*C++ function*), 532  
 xEventGroupClearBitsFromISR (*C macro*), 536  
 xEventGroupCreate (*C++ function*), 529  
 xEventGroupCreateStatic (*C++ function*), 530  
 xEventGroupGetBits (*C macro*), 538  
 xEventGroupGetBitsFromISR (*C++ function*), 536  
 xEventGroupSetBits (*C++ function*), 533  
 xEventGroupSetBitsFromISR (*C macro*), 537  
 xEventGroupSync (*C++ function*), 534  
 xEventGroupWaitBits (*C++ function*), 531  
 xQueueAddToSet (*C++ function*), 482  
 xQueueCreate (*C macro*), 483  
 xQueueCreateSet (*C++ function*), 481  
 xQueueCreateStatic (*C macro*), 484  
 xQueueGenericCreate (*C++ function*), 481  
 xQueueGenericCreateStatic (*C++ function*), 481  
 xQueueGenericReceive (*C++ function*), 478  
 xQueueGenericSend (*C++ function*), 476  
 xQueueGenericSendFromISR (*C++ function*), 475  
 xQueueGiveFromISR (*C++ function*), 476  
 xQueueIsQueueEmptyFromISR (*C++ function*), 476  
 xQueueIsQueueFullFromISR (*C++ function*), 476  
 xQueueOverwrite (*C macro*), 489  
 xQueueOverwriteFromISR (*C macro*), 494  
 xQueuePeek (*C macro*), 490  
 xQueuePeekFromISR (*C++ function*), 477  
 xQueueReceive (*C macro*), 491  
 xQueueReceiveFromISR (*C++ function*), 479  
 xQueueRemoveFromSet (*C++ function*), 483  
 xQueueReset (*C macro*), 496  
 xQueueSelectFromSet (*C++ function*), 483  
 xQueueSelectFromSetFromISR (*C++ function*), 483  
 xQueueSend (*C macro*), 488  
 xQueueSendFromISR (*C macro*), 495  
 xQueueSendToBack (*C macro*), 486  
 xQueueSendToBackFromISR (*C macro*), 493  
 xQueueSendToFront (*C macro*), 485  
 xQueueSendToFrontFromISR (*C macro*), 493  
 xRingbufferAddToQueueSetRead (*C++ function*), 542  
 xRingbufferAddToQueueSetWrite (*C++ function*), 542  
 xRingbufferCreate (*C++ function*), 539  
 xRingbufferCreateNoSplit (*C++ function*), 539  
 xRingbufferGetCurFreeSize (*C++ function*), 539  
 xRingbufferGetMaxItemSize (*C++ function*), 539  
 xRingbufferIsNextItemWrapped (*C++ function*), 539  
 xRingbufferPrintInfo (*C++ function*), 543  
 xRingbufferReceive (*C++ function*), 540  
 xRingbufferReceiveFromISR (*C++ function*), 541  
 xRingbufferReceiveUpTo (*C++ function*), 541  
 xRingbufferReceiveUpToFromISR (*C++ function*), 541  
 xRingbufferRemoveFromQueueSetRead (*C++ function*), 542  
 xRingbufferRemoveFromQueueSetWrite (*C++ function*), 543  
 xRingbufferSend (*C++ function*), 540  
 xRingbufferSendFromISR (*C++ function*), 540  
 xSemaphoreCreateBinary (*C macro*), 497  
 xSemaphoreCreateBinaryStatic (*C macro*), 498  
 xSemaphoreCreateCounting (*C macro*), 508  
 xSemaphoreCreateCountingStatic (*C macro*), 509  
 xSemaphoreCreateMutex (*C macro*), 505  
 xSemaphoreCreateMutexStatic (*C macro*), 506  
 xSemaphoreCreateRecursiveMutex (*C macro*), 507  
 xSemaphoreCreateRecursiveMutexStatic (*C macro*), 507  
 xSemaphoreGetMutexHolder (*C macro*), 511  
 xSemaphoreGive (*C macro*), 501  
 xSemaphoreGiveFromISR (*C macro*), 503  
 xSemaphoreGiveRecursive (*C macro*), 502  
 xSemaphoreTake (*C macro*), 499  
 xSemaphoreTakeFromISR (*C macro*), 505  
 xSemaphoreTakeRecursive (*C macro*), 500  
 xTASK\_SNAPSHOT (*C++ class*), 472  
 xTASK\_SNAPSHOT::pxEndOfStack (*C++ member*), 472  
 xTASK\_SNAPSHOT::pxTCB (*C++ member*), 472  
 xTASK\_SNAPSHOT::pxTopOfStack (*C++ member*), 472  
 xTASK\_STATUS (*C++ class*), 471  
 xTASK\_STATUS::eCurrentState (*C++ member*), 471  
 xTASK\_STATUS::pcTaskName (*C++ member*), 471  
 xTASK\_STATUS::pxStackBase (*C++ member*), 471

xTASK\_STATUS::ulRunTimeCounter (C++ *member*), 471  
 xTASK\_STATUS::usStackHighWaterMark (C++ *member*), 471  
 xTASK\_STATUS::uxBasePriority (C++ *member*), 471  
 xTASK\_STATUS::uxCurrentPriority (C++ *member*), 471  
 xTASK\_STATUS::xHandle (C++ *member*), 471  
 xTASK\_STATUS::xTaskNumber (C++ *member*), 471  
 xTaskCallApplicationTaskHook (C++ *function*), 463  
 xTaskCreate (C++ *function*), 450  
 xTaskCreatePinnedToCore (C++ *function*), 449  
 xTaskCreateStatic (C++ *function*), 452  
 xTaskCreateStaticPinnedToCore (C++ *function*), 451  
 xTaskGetApplicationTaskTag (C++ *function*), 462  
 xTaskGetIdleTaskHandle (C++ *function*), 463  
 xTaskGetIdleTaskHandleForCPU (C++ *function*), 463  
 xTaskGetTickCount (C++ *function*), 460  
 xTaskGetTickCountFromISR (C++ *function*), 461  
 xTaskNotify (C++ *function*), 466  
 xTaskNotifyFromISR (C++ *function*), 467  
 xTaskNotifyGive (C *macro*), 473  
 xTaskNotifyWait (C++ *function*), 468  
 xTaskResumeAll (C++ *function*), 460  
 xTaskResumeFromISR (C++ *function*), 459  
 xTimerChangePeriod (C *macro*), 520  
 xTimerChangePeriodFromISR (C *macro*), 526  
 xTimerCreate (C++ *function*), 511  
 xTimerCreateStatic (C++ *function*), 513  
 xTimerDelete (C *macro*), 522  
 xTimerGetExpiryTime (C++ *function*), 517  
 xTimerGetPeriod (C++ *function*), 517  
 xTimerGetTimerDaemonTaskHandle (C++ *function*), 516  
 xTimerIsTimerActive (C++ *function*), 516  
 xTimerPendFunctionCall (C++ *function*), 518  
 xTimerPendFunctionCallFromISR (C++ *function*), 517  
 xTimerReset (C *macro*), 522  
 xTimerResetFromISR (C *macro*), 527  
 xTimerStart (C *macro*), 519  
 xTimerStartFromISR (C *macro*), 524  
 xTimerStop (C *macro*), 520  
 xTimerStopFromISR (C *macro*), 525